Escola de Camins
Escola Tècnica Superior d'Enginyeria de Camins, Canals i Ports
UPC BARCELONATECH

Shock-capturing in BFECC method on unstructured meshes for Kratos

TREBALL FINAL DE MÀSTER

Treball realitzat per:
Pau Vilar Ribó

Dirigit per:
Pooyan Dadvand
Riccardo Rossi

Màster en:
Enginyeria de Camins, Canals i Ports

Barcelona, 20 de juny de 2016

Departament de Resistència de Materials i Estructures a l'Enginyeria

Shock-capturing in BFECC method on unstructured meshes for Kratos

Author: Pau Vilar Ribó

Thesis Supervisors: Pooyan Dadvand and Riccardo Rossi

June 2016

Abstract

A shock-capturing strategy is presented, deeply studied and implemented in Kratos for solving advection equations on unstructured meshes. The shock-capturing strategy is implemented in the Back and Forth Error Compensation and Correction (BFECC) algorithm, which consists on linearly interpolate the solution in forward and backward sense in order to define a comparative error and improve the accuracy of a simple and unique interpolation without evoking higher order interpolation. The main weakness of the BFECC is the spurious numerical oscillations that the method display in the solution induced by jump discontinuities in the derivatives of the solution as well as by jump discontinuities in the solution itself. The shock-capturing strategy acts as a limiter of the numerical oscillations and requires another backward step in time so that overshoots and undershoots on the new time level are detected when they are moved back to compare with the solution on the old time level. The BFECC method with the implementation of the shock-capturing strategy follow a semi-Lagrangian numerical scheme for the numerical integration of the advection equation discretized on unstructured meshes, which uses an Eulerian framework although the discrete equation come from the Lagrangian perspective.

Keywords: fluid dynamics, advection, pure convection, semi-Lagrangian, BFECC, numerical oscillations, overshooting, undershooting, limiter, shock-capturing.

Acknowledgements

I wish to thank everyone who helped me complete this thesis. Without their continued efforts and support, I would have not been able to bring my work to a successful completion.

I would first like to thank my thesis advisors Pooyan Dadvand and Riccardo Rossi from CIMNE. Their door office was always open whenever I ran into a trouble spot or had a question about my research. They consistently allowed this paper to be my own work, but steered me in the right direction whenever they thought I needed it.

I would also like to acknowledge Carlos Roig from CIMNE. I am gratefully indebted for his very valuable support on the computer implementation part.

Contents

Abstract ..... **v**

Acknowledgements ..... **vii**

CHAPTER 1 Introduction **1**

1.1 Motivation 2
1.2 Layout 3

CHAPTER 2 State of the Art **4**

2.1 Introduction to Flow Simulation 4
2.2 Mathematics of transport phenomena 6
2.2.1 Conservation principles *7*
2.2.2 The generic Transport Equation *8*
2.2.3 Initial and Boundary Conditions *9*
2.3 Hyperbolic transport equations 10
2.4 Discretization of differential operators and variables 11
2.5 Space discretization techniques 12
2.5.1 Computational meshes *13*
2.5.2 Finite Difference methods *14*
2.5.3 Finite Volume methods *15*
2.5.4 Finite Element methods *16*
2.6 Kratos solver 17
2.6.1 Introduction to Kratos *17*
2.6.2 Kratos' structure *18*
2.6.3 Phyton scripts *20*

CHAPTER 3 Objectives **21**

CHAPTER 4 Numerical algorithms and Shock-capturing strategy **22**

4.1 Courant-Friedrichs-Lewy 22
4.2 The Upwind scheme 24
4.3 Numerical approach: Semi-Lagrangian scheme 27
4.4 Back and Forth Error Compensation and Correction 29
4.4.1 Forward advection *29*
4.4.2 Backward advection *30*
4.4.3 Error Compensation and Correction *31*
4.5 Numerical oscillations in sharp fronts 32
4.6 BFECC with Shock-capturing strategy 33
4.6.1 Backward advection to define comparative error *33*
4.6.2 Limiting *34*

4.6.3 Forward with modified solution 34

CHAPTER 5 Comparison analysis of resolution methods 35

5.1 Numerical solution for linear advection equation in 1D 37
5.1.1 1D Gaussian pulse 37
5.1.2 1D Pyramid 40
5.1.3 1D Square wave 42
5.2 Numerical solution for advection equation in 2D 45
5.2.1 2D Gaussian pulse 46
5.2.2 2D Square wave 49
5.3 Kratos numerical test example in 2D 53

CHAPTER 6 Computer implementation 58

6.1 Algorithm implementation in Kratos 58
6.2 Interpolation strategy 60
6.3 BFECC with shock-capturing strategy routine in Kratos 61
6.3.1 Main routine 61
6.3.2 Complementary functions 68
6.4 One step further: application of the shock-capturing strategy at the barycentre of the elements 70

CHAPTER 7 Correction, validation and examples 74

7.1 Improving the numerical solution for linear advection equation in 1D 74
7.1.1 1D Gaussian pulse 74
7.1.2 1D Pyramid 77
7.1.3 1D Square wave 79
7.2 Kratos numerical test example in 2D with shock-capturing strategy 82

CHAPTER 8 Conclusions 86

APPENDIX A. Matlab codes for solving numerical test examples 89

A.1. 1DConvection.m 89
A.2. f_piecewise() 92
A.3. f_pyramid() 92
A.4. 2DConvection.m 93
A.5. gaussian2D() 96
A.6. cosHump2D() 96
A.7. minmod() 97

References 99

Consulted literature 101

List of Figures

Figure 1. Control volume V bounded by the control surface S 8
Figure 2. Schematic discretization of an hyperbolic initial value problem 12
Figure 3. Structured and uniform mesh 14
Figure 4. Unstructured meshes, made by triangular (left) and quadrilateral elements (right) 14
Figure 5. Control volume for a cell-centred FVM in 2D, in a structured (left) and an unstructured mesh (right) 15
Figure 6. Control volume for a vertex-centred FVM in 2D, in a structured (left) and an unstructured mesh (right) 16
Figure 7. Kratos in the modelling process 18
Figure 8. Kratos object-oriented structure 19
Figure 9. Triangular property distribution 26
Figure 10. Initial position of φ at time step tn 29
Figure 11. Position of the particle at time step $tn + 1$ after moving it forwards 30
Figure 12. Position of the particle at time step tn after moving it backwards 31
Figure 13. Error between the initial and the moved position 31
Figure 14. Correction of the particle's position at time step tn 32
Figure 15. Particle's final position at $tn + 1$ after moving the corrected position forward 32
Figure 16. Periodic boundary conditions scheme 37
Figure 17. 1D Gaussian pulse advection with CFL = 0.5 38
Figure 18. 1D Gaussian pulse advection with CFL = 1.0 39
Figure 19. 1D Gaussian pulse advection with CFL = 2.0 39
Figure 20. 1D Pyramid advection with CFL = 0.5 40
Figure 21. 1D Pyramid advection with CFL = 1.0 41
Figure 22. 1D Pyramid advection with CFL = 2.0 42
Figure 23. 1D Square wave advection with CFL = 0.5 43
Figure 24. 1D Square wave advection with CFL = 1.0 44
Figure 25. 1D Square wave advection with CFL = 2.0 45
Figure 26. 2D Gaussian pulse advected by Upwind method 47
Figure 27. 2D Gaussian pulse advected by BFECC method 49
Figure 28. 2D Square wave advected by Upwind method 50
Figure 29. 2D Square wave advected by BFECC method 52
Figure 30. Initial condition for Kratos test example 53
Figure 31. Velocity field for Kratos test example 54
Figure 32. Velocity field's code for Kratos test example 54
Figure 33. Kratos' solution with BFECC method after one loop 55
Figure 34. Numerical oscillations with BFECC method after one loop 56
Figure 35. Kratos' solution with BFECC method at final time step 56
Figure 36. Numerical oscillations with BFECC method at final time step 57
Figure 37. Bilinear interpolation 60
Figure 38. Trilinear interpolation 61
Figure 39. Processing information from the *ModelPart* 62

Figure 40. Backward advection 63
Figure 41. Forward advection 64
Figure 42. Computing the error of the BFECC method 64
Figure 43. Backward advection with modified solution 65
Figure 44. Computing the comparative error 66
Figure 45. Finding the nodes of the neighbour elements 66
Figure 46. *minmod* limiter where numerical oscillations occur 67
Figure 47. New correction of the solution at time step n 67
Figure 48. Backward advection with corrected solution 68
Figure 49. *minmod* function 68
Figure 50. *ConvectBySubstepping* function 70
Figure 51. Limiting strategy at elements nodes 71
Figure 52. Limiting strategy at elements barycentre 71
Figure 53. Defining variables in the element's barycentre 72
Figure 54. Computing the comparative error in the barycentre 72
Figure 55. *minmod* limiter in the same element 73
Figure 56. Correcting nodal errors from each element 73
Figure 57. Correction of 1D Gaussian pulse advection with CFL = 0.5 75
Figure 58. Correction of 1D Gaussian pulse advection with CFL = 1.0 75
Figure 59. Correction of 1D Gaussian pulse advection with CFL = 2.0 76
Figure 60. Correction of 1D Pyramid advection with CFL = 0.5 77
Figure 61. Correction of 1D Pyramid advection with CFL = 1.0 78
Figure 62. Correction of 1D Pyramid advection with CFL = 2.0 78
Figure 63. Correction of 1D Square wave advection with CFL = 0.5 79
Figure 64. Correction of 1D Square wave advection with CFL = 1.0 80
Figure 65. Correction of 1D Square wave advection with CFL = 2.0 81
Figure 66. Comparison of final numerical test example' solution 83
Figure 67. Comparison of final numerical test example' solution 84
Figure 68. Numerical oscillations of shock-capturing at the barycentre 85

List of Tables

Table 1. 1D Gaussian pulse numerical oscillations with CFL = 0.5 38
Table 2. 1D Gaussian pulse numerical oscillations with CFL = 1.0 39
Table 3. 1D Gaussian pulse numerical oscillations with CFL = 2.0 40
Table 4. 1D Pyramid numerical oscillations with CFL = 0.5 40
Table 5. 1D Pyramid numerical oscillations with CFL = 1.0 41
Table 6. 1D Pyramid numerical oscillations with CFL = 2.0 42
Table 7. 1D Square wave numerical oscillations with CFL = 0.5 43
Table 8. 1D Square wave numerical oscillations with CFL = 1.0 44
Table 9. 1D Square wave numerical oscillations with CFL = 2.0 45
Table 10. 2D Gaussian pulse advected by Upwind method 46
Table 11. 2D Gaussian pulse advected by BFECC method 48
Table 12. 2D Square wave advected by Upwind method 49
Table 13. 2D Square wave advected by BFECC method 51
Table 14. Numerical solution after one loop 55
Table 15. Numerical oscillations at final time step 56
Table 16. Algorithm variable's name in Kratos code 59
Table 17. 1D Gaussian pulse numerical oscillations correction with CFL = 0.5 75
Table 18. 1D Gaussian pulse numerical oscillations correction with CFL = 1.0 76
Table 19. 1D Gaussian pulse numerical oscillations correction with CFL = 2.0 76
Table 20. 1D Pyramid numerical oscillations correction with CFL = 0.5 77
Table 21. 1D Pyramid numerical oscillations correction with CFL = 1.0 78
Table 22. 1D Pyramid numerical oscillations correction with CFL = 2.0 79
Table 23. 1D Square wave numerical oscillations correction with CFL = 0.5 80
Table 24. 1D Square wave numerical oscillations correction with CFL = 1.0 81
Table 25. 1D Square wave numerical oscillations correction with CFL = 2.0 81
Table 26. BFECC with shock-capturing strategy solution 83
Table 27. BFECC with shock-capturing strategy at the element's barycentre 85

CHAPTER 1

Introduction

Convection-diffusion transport problems often appear in science and engineering areas. Some applications of these kind of problems can be found when dealing with transport of air and ground water pollutants, oil reservoir flow, convective heat transfer, atmospheric circulation, weather, oceanic circulation, among other examples. There are specific sorts of physical transport phenomena where diffusion effect does not play any role. These type of problems are called pure advection problems where advection refers to the transport mechanism of a substance or conserved property by a fluid due to the fluid's motion. A simple example of advection is the transport of pollutants or granular materials in a river by bulk water flow downstream, see [1] and [2] for further reading on fluid dynamics concepts.

Advection problems are governed by partial differential equations (PDE) [7] that describe the motion of a conserved scalar field (transported material) as it is advected by a velocity field (fluid's motion). The solution of the advection equation cannot be found analytically, apart from special cases such as when the equation is a linear first order PDE, but may be found by using numerical methods. Many are the numerical methods used to solve advection problems. Typically they are classified in three main techniques: Finite Volume Methods (FVM), Finite Difference Methods (FDM) and Finite Elements Methods (FEM). This report will focus on finite difference and finite elements methods, specifically in the application of the Back and Forth Error Compensation and Correction (BFECC) method. The BFECC method is essentially a strategy for finding the error committed on the forward interpolation for finding the solutions in the next time step. The technique consists on finding again the solution backward in time and evaluate the difference between the two copies of the solution at

the initial time level. This difference provides information about the numerical error of the interpolation scheme used to advect the particles.

Even though the simulation of advection problems may be performed with more or less accuracy, the hyperbolic partial differential equation that governs advection problems might be complex to solve when there exists sharp fronts or jump discontinuities in the derivatives of the solution as well as in the solution itself. The non-smooth characteristic of the problem may lead to numerical oscillations on the final solution, which are spurious and make no physical sense.

The Shock-capturing strategy presented in this essay aims to improve the BFECC strategy not only correcting the error committed while interpolating but also eliminating the numerical oscillations performed by the method applied. Particularly, the shock-capturing strategy requires another interpolation backward in time to compute another comparative error that will be used to identify if the solution of the original BFECC method conduct to overshoots or undershoots, which refer to the numerical oscillations.

The shock-capturing strategy will be implemented in Kratos, which is a framework developed in CIMNE [8] for the implementation of numerical methods for the solution of engineering problems. Among other applications, Kratos is able to solve advection problems with the convection-diffusion application. It currently solves transport problems by finite elements, but it is continuously being improved, since it is an open-source solver. This essay takes part on a project which deals with the implementation of the BFECC method in Kratos, more specifically, the objective of this work is to implement the shock-capturing strategy in the BFECC scheme.

1.1 Motivation

So many progress have been done in computational engineering the last decades, but there is still much work to do. Computers are constantly increasing their capacity and power, so better simulations of physical phenomena can be made by using finer meshes or solving trickiest problems with non-linear behaviour for example. The complexities of simulations for advection problems are related with the discretization of the domain and the approximation of the space derivatives. So better mathematical algorithms and strategies need to be developed and implemented in our solvers. This report deals with the implementation of a recently developed strategy for improving the simulation of advection equations.

Kratos is a solver partially developed in CIMNE [8], which is a research centre placed in front of the international investigation on computational engineering, so collaborating with the development of research projects in CIMNE is always challenging and encouraging.

1.2 Layout

With the purpose of making the reading clear and understandable, the report have been structured in the following chapters:

CHAPTER 1. Introduction. The topic of the essay is presented as well as the context in which is placed in. A general view of the physical problem, the numerical methods and the computer solver used is explained.

CHAPTER 2. State of the Art. Contains some mathematical basic concepts as well as the main numerical methods used to solve transport problems to be kept in mind in order to follow the succeeding reading. Besides, a general description of the current modified solver, Kratos, is made. How Kratos works and which programming languages and files are needed to solve problems will be treated in this chapter.

CHAPTER 3. Objectives. There are list the main objectives and purposes of the essay together with the derived goals.

CHAPTER 4. Numerical algorithms and Shock-capturing strategy. This is the main chapter where the BFECC method and the shock-capturing strategy will be explained in detail to be understood before dealing with the computer implementation.

CHAPTER 5. Comparison analysis of resolution methods. Many numerical tests examples will be performed and analysed in this chapter in order to identify the strengths and weaknesses of the different methods where numerical oscillations may appear and how they behave in front of different functions and conditions.

CHAPTER 6. Computer implementation. The implementation of the shock-capturing strategy for the BFECC method in Kratos is detailed in this section. The implemented code will be explained step by step.

CHAPTER 7. Correction, validation and examples. In this chapter it is going to be shown whether the implementation of the shock-capturing strategy works or not and how it works. Many numerical test examples will be solved in order to analyse when and where this strategy is eliminating the spurious oscillations.

CHAPTER 8. Conclusions. A summary of the main outcomes of the essay will be written in this chapter, identifying the success or the failure of the implementation of the shock-capturing strategy for solving advection equations in Kratos.

CHAPTER 2

State of the Art

The aim of this chapter is to overview the state of knowledge in the field where the essay is placed in. This section contains basic background information about flow simulation modelling, more specifically the numerical techniques that are used to solve advection equations. It also includes a general outlook of the main strategies that are implemented in the solvers in order to solve partial differential equations, such as space discretization techniques and time-stepping techniques. Finally, there is a section which aims to briefly comment what is Kratos, which structure defines it and how it works.

2.1 Introduction to Flow Simulation

Fluid dynamics and transport phenomena, such as heat and mass transfer, play a vitally important role in human life. Gases and liquids surround us and flow inside our bodies have a profound influence on the environment in which we live. Fluid flows produce winds, rains, floods, and hurricanes. Convection and diffusion are responsible for temperatu re fluctuations and transport of pollutants in air, water or soil. The ability to understand, predict, and control transport phenomena is essential for many industrial applications, such as aerodynamic shape design, oil recovery from an underground reservoir, or multiphase/multicomponent flows in furnaces, heat exchangers, and chemical reactors. This ability offers substantial economic benefits and contributes to human well-being. Heating, air conditioning, and weather forecast have become an integral part of our everyday life. Such things are usually taken for granted and hardly ever think about the physics and mathematics behind them.

The traditional approach to investigation of a physical process is based on observations, experiments, and measurements. The amount of information that can be obtained in this way is usually very limited and subject to measurement errors. Moreover, experiments are only possible when a small-scale model or the actual equipment has already been built. An experimental investigation may be very time-consuming, dangerous, prohibitively expensive, or impossible for another reason.

Alternatively, an analytical or computational study can be performed on the basis of a suitable mathematical model. As a rule, such a model consists of several differential and/or algebraic equations which make it possible to predict how the quantities of interest evolve and interact with one another. A drawback to this approach is the fact that complex physical phenomena give rise to complex mathematical equations that cannot be solved analytically, i.e., using paper and pencil.

The most detailed models of fluid flow are based on *first principles*, such as the conservation of mass, momentum, and energy. Mathematical equations that embody these fundamental principles have been known for a very long time but used to be practically worthless until numerical methods and digital computers were invented. The second half of the twentieth century has witnessed the advent of Computational Fluid Dynamics (CFD), see [3] and [4], a new branch of applied mathematics that deals with numerical simulation of fluid flows. Nowadays, computer codes based on CFD models are used routinely to predict a variety of increasingly complex flow phenomena.

The quality of simulation results depends on the choice of the model and on the accuracy of the numerical method. In spite of the inevitable numerical and modelling errors, approximate solutions may provide a lot of valuable information at a fraction of the cost that a full-scale experimental investigation would require. Moreover, the sampling of relevant data is free of errors due to a flow disturbance caused by probes. A further advantage of the computational approach is the fact that it can be applied to flows in domains with arbitrarily large or small dimensions under realistic operating conditions. High pressures, toxic chemicals or hot temperatures pose no hazard to a CFD practitioner. Last but not least, simultaneous computation of instantaneous density, velocity, pressure, temperature, and concentration fields is feasible. Clearly, no experimental technique can capture the evolution of all flow variables throughout the domain. However, experiments are still required to determine the values of input parameters for a mathematical model and to validate the computational results.

The choice of a CFD model is dictated by the nature of the physical process to be simulated, by the objectives of the numerical study, and by the available resources. As a rule of thumb, the mathematical model should be as detailed as possible without making the computations too expensive. The use of a universally applicable model makes it difficult to develop and implement an efficient numerical algorithm. In many cases, the desired information can be obtained using a simplified version that exploits some a priori knowledge of the flow pattern or incorporates empirical correlations

supported by theoretical or experimental studies. Thus, a hierarchy of fundamental, phenomenological, and empirical models is usually available for particularly difficult problems, such as the numerical simulation of turbulence.

Over the past three decades, the market for CFD software has expanded rapidly, and remarkable progress has been made in the development of numerical algorithms. An astonishing variety of finite difference, finite element, finite volume, and spectral schemes were developed for the equations of fluid mechanics and applied to virtually every flow problem of practical importance. Modern CFD codes are equipped with automatic mesh generation/adaptation tools, such as GiD [5] for Kratos solver, reliable error control mechanisms, and efficient iterative solvers for sparse linear systems. Unstructured mesh methods are available for flows in complex geometries. Problems with moving boundaries and free interfaces can be solved in a fixed or moving reference frame. Parallelization and vectorization make it possible to perform large-scale computations with more than a billion of degrees of freedom. The rapid growth of computing power has stimulated implementation of sophisticated models and extended the range of possible applications to problems as complex as turbulent multiphase flows and fluid-structure interaction. Nowadays, 3D simulations of unsteady transport processes can be performed on a laptop or desktop computer, whereas supercomputers were required to simulate steady 2D problems a couple of decades ago.

If something sounds too good to be true, it probably is. In spite of the abovementioned recent advances, there is still a lot of room of improvement when it comes to reliable simulation of transport phenomena. The user of a commercial CFD code might be unaware of the numerous subtleties, trade-offs, compromises, and ad hoc tricks involved in the computation of beautiful colourful pictures. Usually, there is no guarantee that these pictures are quantitatively correct. If the same problem is solved using another mesh, another time step, and/or another numerical scheme, then a qualitatively different solution may be obtained. Hence, the results of a CFD simulation should not be taken at their face value even if they look 'nice' and plausible. In other cases, the approximate solution may exhibit spurious oscillations and/or assume nonphysical negative values. This behaviour is typical of problems with discontinuities and steep fronts that cannot be resolved properly on a given mesh. Therefore, it might be necessary to refine the mesh and/or adjust the coefficients of the numerical scheme if nonphysical solution behaviour is detected. Ideally, the numerical algorithm should do it automatically by adapting itself to the nature of the problem at hand so as to compute accurate solutions in an efficient way.

2.2 Mathematics of transport phenomena

The derivation of differential equations that govern the evolution of scalar fluid properties is usually based on certain conservation principles, as applied to an arbitrary *control volume* $V \subset \mathbb{R}^d$, where d

= 1, 2, or 3 is the number of space dimensions. If the fluid is in motion, it may flow in and out across the *control surface* S which forms the boundary of V, see Figure 1. Individual molecules may travel across the interface even if the fluid is at rest. Therefore, the physical and chemical properties of the fluid inside V are influenced by those of the surrounding medium. Moreover, some quantities, such as mass, momentum, and energy, are *conserved*. That is, they may move from one place to another but cannot emerge out of nothing or disappear spontaneously. The physical forces that transport, produce or destroy these quantities are well-known, and reliable mathematical models are available. Thus, conservation principles can be expressed in terms of differential equations that describe all relevant transport mechanisms, such as *convection* (also called *advection*), *diffusion*, and *dispersion*.

2.2.1 Conservation principles

Let $c(\boldsymbol{x}, t) \in \mathbb{R}$ denote the concentration (amount per unit mass) of a scalar conserved quantity at point $\boldsymbol{x} \in V$ and time $t \geq 0$. The corresponding concentration per unit volume is given by $u = \rho c$, where ρ is the density of the carrier fluid. The total amount of the conserved variable inside V is given by the volume integral

$$\int_V u(\boldsymbol{x}, t)\, d\boldsymbol{x} = \int_V \rho(\boldsymbol{x}, t) a(\boldsymbol{x}, t)\, d\boldsymbol{x} \tag{1}$$

This integral is called the *mass* and a can represent either the energy, a single velocity component or another dimensional quantity.

Obviously, the variation of (1) depends on the rate at which c enters or leaves V through the boundary S. This rate is called the *flux* and denoted by

$$\boldsymbol{f}(\boldsymbol{x}, t) = (f^1, \dots, f^d) \tag{2}$$

where f^k corresponds to the rate of transport in the k-th coordinate direction, per unit area and time. If $\boldsymbol{ds} = \boldsymbol{n}\, ds$ is an infinitesimally small patch of S with the unit outward normal $\boldsymbol{n}$, then the mass crossing this patch per unit time is $\boldsymbol{f} \cdot \boldsymbol{n}\, ds$.

In the simplest case, the flux vector $\boldsymbol{f}$ is a linear function of u and/or $\rho \nabla a$, where

$$\nabla = \left(\frac{\partial}{\partial x_1}, \dots, \frac{\partial}{\partial x_d} \right)^T \tag{3}$$

is the vector of partial derivatives that defines the gradient and divergence operators.

Chemical reactions, heating cooling, and similar processes give rise to interior sources or sinks that generate $s(\boldsymbol{x}, t)$ units of mass per unit volume and time. Thus, the temporal variation of (1) satisfies an integral conservation law of the form

$$\frac{\partial}{\partial t}\int_V u(\boldsymbol{x},t)\,d\boldsymbol{x} + \int_S \boldsymbol{f}\cdot\boldsymbol{n}\,ds = \int_V s(\boldsymbol{x},t)\,d\boldsymbol{x} \tag{4}$$

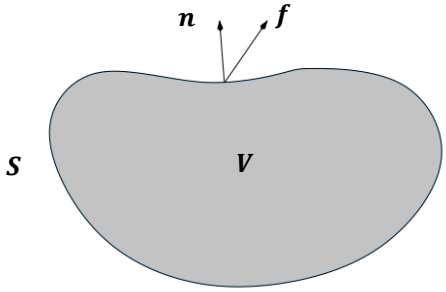


Figure 1. Control volume V bounded by the control surface S

The surface integral is the mass that leaves V per unit area and time, whereas the right-hand side of (4) corresponds to the mass produced inside V per unit time.

If the functions $\boldsymbol{u}(\boldsymbol{x},t)$ and $\boldsymbol{f}(\boldsymbol{x},t)$ are differentiable, then the divergence theorem [6], as applied to the surface integral in (4), yields the identity

$$\int_V \left[\frac{\partial u(\boldsymbol{x},t)}{\partial t} + \nabla\cdot\boldsymbol{f}(\boldsymbol{x},t) - s(\boldsymbol{x},t)\right] d\boldsymbol{x} = 0 \tag{5}$$

Since the choice of V is arbitrary, the expression in the square brackets must vanish, so the evolution of $\boldsymbol{u}(\boldsymbol{x},t)$ is governed by the following partial differential equation

$$\frac{\partial u(\boldsymbol{x},t)}{\partial t} + \nabla\cdot\boldsymbol{f}(\boldsymbol{x},t) = s(\boldsymbol{x},t) \tag{6}$$

If the divergence theorem is applicable, this *differential form* of the conservation law is equivalent to the underlying *integral form* (4). However, the latter is more fundamental since it does not contain any space derivatives. If the flux $\boldsymbol{f}(\boldsymbol{x},t)$ does not depend on the gradients of $\boldsymbol{c}$, then the generalized solution may exhibit very steep gradients or even discontinuities. Such solutions satisfy (4) but not (6) since the discontinuous functions are not differentiable in the classical sense.

2.2.2 The generic Transport Equation

The *generic transport equation* has the following form:

$$\frac{\partial \rho c}{\partial t} + \nabla\cdot(\boldsymbol{a}\rho c) - \nabla\cdot(\mathcal{D}\rho\nabla c) = s \tag{7}$$

The terms that appear in this equation admit the following physical interpretation

- the rate-of-change term $\frac{\partial \rho c}{\partial t}$ is the net gain/loss of mass per unit volume and time;
- the convective term $\nabla\cdot(\boldsymbol{a}\rho c)$ is due to the downstream transport with velocity $\boldsymbol{a}$;

- the diffusive term $-\nabla \cdot (\mathcal{D}\rho\nabla c)$ is due to a non-uniform spatial distribution of c;
- the source or sink term s combines all other effects that create or destroy ρc.

For the time being, it is assumed that the parameters ρ, $\boldsymbol{a}$, $\mathcal{D}$ and s are known. In reallife applications, they may depend on the concentration c and/or other variables.

In particular, conservation laws of the form (7) constitute the *Navier-Stokes equations*, in which the conserved variables are the mass, momentum, and total energy. The simplest component of this PDE system is the *continuity equation*

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\boldsymbol{a}\rho) = 0 \tag{8}$$

which is responsible for mass conservation and corresponds to (7) with $c = 1$ and s= 0. Note that the diffusive term vanishes since the gradient of c is zero. If viscosity and heat conduction are neglected, then the Navier-Stokes equations reduce to the *Euler equations* that describe inviscid gas flows at high speeds.

The common structure of mathematical models which are based on (systems of) scalar conservation laws of the form (7) suggests a systematic approach to analysis, discretization, and coding. This strategy facilitates the development, implementation, and testing of numerical methods for advanced CFD applications. In addition to the conceptual and algorithmic simplicity, it offers a simple way to investigate the solution behaviour in important limiting cases (steady state, pure convection, pure diffusion etc.) and design simple test problems that can be solved analytically.

2.2.3 Initial and Boundary Conditions

A differential equation governs a family of solutions. A particular member of the family of solutions is specified by the auxiliary conditions imposed on the differential equation. For steady-state equilibrium problems, the auxiliary conditions consist of boundary conditions on the entire boundary on the entire boundary of the closed solution domain. Three types of boundary conditions can be imposed:

- *Dirichlet boundary condition*: The value of the unknown function is specified on the boundary.
- *Neumann boundary condition*: The value of the derivative normal to the boundary is specified on the boundary, as a normal flux.
- *Mixed boundary condition*: A combination of the unknown function and its derivative normal to the boundary is specified on the boundary.

One of the above types of boundary conditions must be specified at each point on the boundary of the closed solution domain. Different types of boundary conditions can be specified on different portions of the boundary.

For unsteady or steady propagation problems, the auxiliary conditions consist of an initial condition along the time boundary that defines the distribution of the mass at $t = 0$ and boundary conditions on the physical boundaries of the solution domain. No auxiliary conditions can be applied on the open boundary in the time direction. For a PDE containing a first-order time derivative, one initial condition is required along the time boundary. For a PDE containing second-order time derivatives, two initial conditions are required along the time boundary.

2.3 Hyperbolic transport equations

The purely convective transport problem is presented in this section. If no diffusion is considered in a steady problem, that is for $\mathcal{D} = 0$, the general transport equation degenerates into a *hyperbolic* PDE of first order

$$\nabla \cdot (\boldsymbol{a}u) = s \tag{9}$$

In this case, information is transported at finite speeds along the streamlines of the stationary velocity field $\boldsymbol{a}(\boldsymbol{x})$.

The unsteady version of the convection-reaction equation (9) is given by

$$\frac{\partial u}{\partial t} + \nabla \cdot (\boldsymbol{a}u) = 0 \tag{10}$$

The most prominent example is the continuity equation (8) with $u = \rho$ and $s = 0$, that is the *pure advection equation*.

In the one-dimensional case with constant velocity, the equations has the following form:

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0 \tag{11}$$

Like any other PDE of first order, equation (10) is of hyperbolic type. The direction and speed of convective transport depend on the velocity field $\boldsymbol{a}(\boldsymbol{x}, t)$. The initial condition is given by

$$u(\boldsymbol{x}, 0) = u_0(\boldsymbol{x}) \tag{12}$$

and information travels forward in time. That is, the distribution of u at any time instant $\bar{t}$ depends on the previous evolution history but only the solution at a later time may be influenced by what happens at $t = \bar{t}$.

Analytical solutions to (9) and (10) can be constructed by the *method of characteristics* [7]. Due to the lack of diffusive effects, hyperbolic conservation laws admit discontinuous and, possibly, non-unique weak solutions. Such problems are particularly difficult to solve numerically, although a lot of information about the properties of exact solutions is available.

2.4 Discretization of differential operators and variables

A simple generic one-dimensional differential equation can be written as

$$\mathcal{L}(u) - f = 0 \tag{13}$$

where $u = u(x,t)$ and $\mathcal{L}$ is a differential operator in the two variables x and t acting on u. One of the most used methods for the solution of such problem is by means of *finite differences*, explained in more detail in posterior sections. It consists of two "discretization steps":

- *Variables discretization*: replace the function $u(x,t)$ with a discrete set of values $\{u_j^n\}$ that should approximate the pointwise values of u, that is $u_j^n \approx u(x_j, t^n)$.
- *Operator discretization*: replace the continuous differential operator $\mathcal{L}$ with a discretized one, $\mathcal{L}_\Delta$, that when applied to the set $\{u_j^n\}$, gives an approximation to $\mathcal{L}(u)$ in terms of differences between the various u_j^n.

So, it is useful to discretize the continuum space of solution, space-time in the case of hyperbolic non-steady equations, in spatial foliations such that the time coordinate t is constant on each slice. The space discretization is done generating computational meshes, as will be developed in following sections. As shown in Figure 2, each point (x_j, t^n) in this discretized space-time will have spatial and time coordinate define as

$$\begin{aligned} x_j &= x_0 + j\Delta x, \qquad & j &= 0, \pm 1, \dots, \pm J \\ t^n &= t^0 + n\Delta t, \qquad & n &= 0, \pm 1, \dots, \pm N \end{aligned} \tag{14}$$

where Δx and Δt are the increments between two spacelike and timelike foliations, respectively. Clearly, the number of discrete solutions to be associated to $u(x,t)$ will depend on the properties of the discretised space-time (i.e., on the increments Δx and Δt) which will also determine the *truncation error* [19] introduced by the discretization.

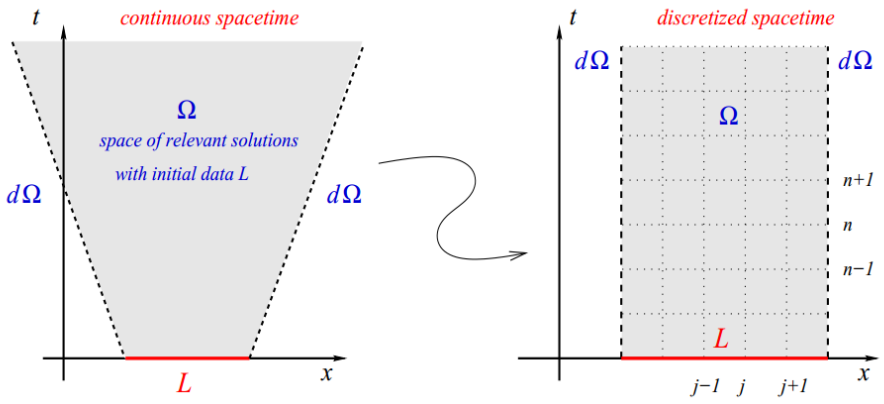


Figure 2. Schematic discretization of an hyperbolic initial value problem

2.5 Space discretization techniques

Computers are of little help in obtaining closed-form analytical solutions to a PDE model. However, they can be programmed to solve algebraic equations very fast. Replacing calculus by algebra, it is possible to compute approximate solutions to the Convection-Diffusion-Reaction (CDR) equation and more advanced mathematical models. To this end, the computational domain, the unknown solution, and its partial derivatives need to be discretized, so as to obtain a set of algebraic equations for the function values at a finite number of discrete locations.

Partial differential equations cannot be solved analytically for general cases. So, numerical methods are used to solve PDEs with the computer. The three most widely used numerical methods are the Finite Elements Method (FEM), Finite Volume Methods (FVM) and Finite Difference Methods (FDM). The FEM is a numerical technique for finding approximate solutions of PDEs as well as of integral equations. The solution approach is based either on eliminating the differential equation completely (steady-state problems) or rendering the PDE into an approximating system of ordinary differential equations (ODEs), which are then numerically integrated using standard techniques such as Euler's method, Runge Kutta, etc. The FDM are numerical methods for approximating the solutions to differential equations using finite difference equations to approximate derivatives in space and time. Finally, in the FVM values are calculated at discrete places on a meshed geometry. *Finite volume* refers to the small volume surrounding each node point on a mesh. In the finite volume method, surface integrals in a partial differential equation that contain a divergence term are converted to volume integrals, using the *Divergence theorem* [10]. These terms are then evaluated as fluxes at the surfaces of each finite volume. Because the flux entering a given volume is identical to that leaving the adjacent volume, these methods are conservative.

2.5.1 Computational meshes

Recall that the integral conservation law (4) which has led to (7) was formulated for a fixed control volume of finite size, see Figure 1. Instead of looking at the whole flow field at once, the attention has to be focused on what is happening in a small subdomain. A similar approach is used to discretize differential equations that embody physical conservation principles. The unknowns of the discrete problem are associated with a computational *mesh* or *grid* which represents a subdivision of the domain into small control volumes, for example intervals in 1D, triangles or quadrilaterals in 2D or tetrahedral in 3D.

Many references about computational meshes can be found in [13] [14] and [15]. Mesh generation is easy for domains of rectangular shape but difficult in the case of curvilinear boundaries, internal obstacles and small-scale features. Depending on the geometric complexity of the domain the mesh may be structured on unstructured.

In the one-dimensional case, the computational domain is an interval (a, b). A subdivision of this interval into N subintervals of equal size

$$\Delta x = \frac{b - a}{N}$$

yields the simplest representative of structured meshes. The $N + 1$ grid points

$$x_i = i\Delta x, \qquad \forall i = 0,1, \dots, N \tag{15}$$

are numbered from left to right. Each interior grid point x_i has two nearest neighbours whose indices $i \pm 1$ and coordinates $x_{i\pm1}$ are known. The spacing $\Delta x_k = x_k - x_{k-1}$ can also be non-uniform if higher mesh resolution is desired in some regions.

In multidimensional problems, a structured mesh is a net of grid lines which can be numbered consecutively. Again, the search of the nearest neighbours is easy, and their number is the same for each interior point.

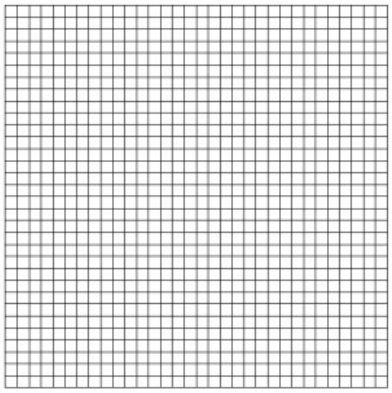


Figure 3. Structured and uniform mesh

Domains of particularly complex geometric shape call for the use of fully unstructured mesh. Recent advances in computational geometry make it possible to generate such meshes automatically for 2D and 3D problems, see [5]. Unstructured mesh methods are very flexible and well suited for mesh adaptivity. An arbitrary number of elements are allowed to meet at a single vertex. So it is easy to insert extra grid points in regions of insufficient mesh resolution.

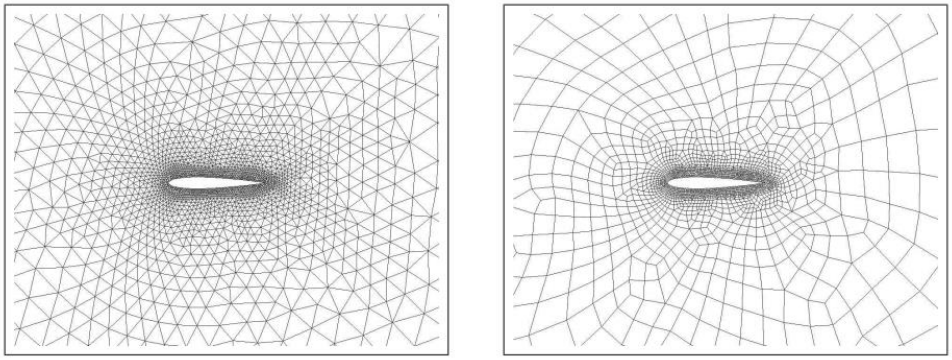


Figure 4. Unstructured meshes, made by triangular (left) and quadrilateral elements (right)

2.5.2 Finite Difference methods

The finite difference method (FDM) is the oldest among the discretization techniques for partial differential equations. Many modern numerical schemes for transport phenomena trace their origins to finite difference approximations developed in the late 1950s through early 1980s. The derivation and implementation of FDM are particularly simple on structured meshes which are topologically equivalent to a uniform Cartesian grid. The nodal value of the approximate solution at node i

$$u_i(t) \approx u(\boldsymbol{x}_i, t) \tag{16}$$

Is a pointwise approximation to the true solution of the partial differential equation.

Taylor series expansions or polynomial fitting techniques are used to approximate all space derivatives in terms of u_i and/or solution values at a number of neighbouring nodes. For example, if it considered a the uniform 1D mesh given by (15), then

$$\left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_{i+1} - u_{i-1}}{2\Delta x} \tag{17}$$

is a second-order approximation to the first derivative of u at node i, whereas

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_i \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2} \tag{18}$$

is a second-order approximation to the second derivative. On a non-uniform mesh, the coefficients are different and must be derived individually for each grid point.

2.5.3 Finite Volume methods

Due to the growing demand for numerical simulation of transport processes in 2D and 3D domains of complex shape, the finite difference method has eventually lost its leadership position. Nowadays, general-purpose CFD codes are typically based on the finite volume method (FVM) which yields a finite-difference like approximation on a uniform Cartesian grid but is readily applicable to unstructured meshes.

Finite volume methods for the CDR equation (7) are based on the underlying integral conservation law. Inserting the flux $\boldsymbol{f} = \boldsymbol{a}u - \mathcal{D}\nabla u$ into (4), one obtains

$$\frac{\partial}{\partial t}\int_{V_i} u(\boldsymbol{x},t)\, d\boldsymbol{x} + \int_{S_i} (\boldsymbol{a}u - \mathcal{D}\nabla u)\cdot \boldsymbol{n}\, ds = \int_{V_i} s(\boldsymbol{x},t)\, d\boldsymbol{x} \tag{19}$$

where V_i is a control volume bounded by the control surface S_i, and $\boldsymbol{n}$ is the unit outward normal. In *cell-centred* finite volume methods, the control volume is a single cell of the computational mesh, see Figure 5.

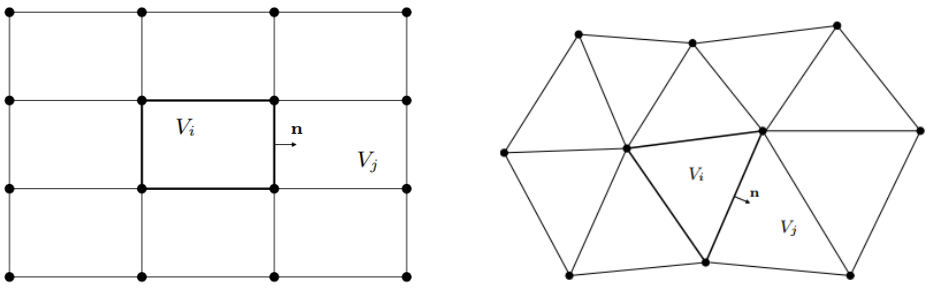


Figure 5. Control volume for a cell-centred FVM in 2D, in a structured (left) and an unstructured mesh (right)

Alternatively, a dual tessellation can be used to define V_i for a *vertex-centred* finite volume method. In the two-dimensional case, the dual cell V_i around the vertex $\boldsymbol{x}_i$ can be constructed by joining the midpoints of mesh edges and the centroids of the neighbouring cells, as shown in

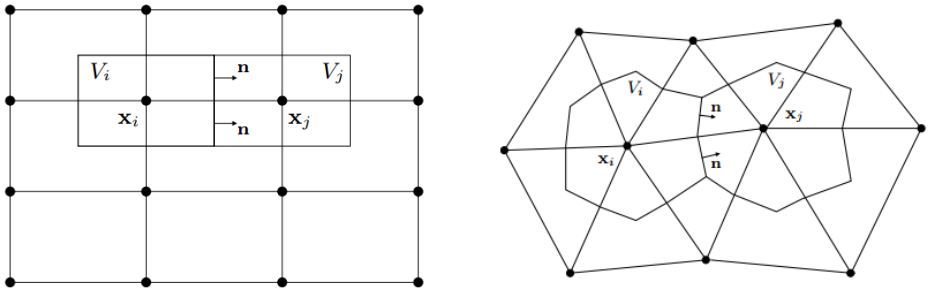


Figure 6. Control volume for a vertex-centred FVM in 2D, in a structured (left) and an unstructured mesh (right)

This method leads to expressions with surface and volume integrals which are approximated using numerical quadrature, that require the evaluation of the integrand at one or more locations. Another approximation is made in relation to the function values and its derivatives. To do so, interpolation techniques are employed at the quadrature points in terms of the primary unknowns $\boldsymbol{u}_i$.

2.5.4 Finite Element methods

The finite element method (FEM) is a relative newcomer to CFD and a very promising alternative to finite differences and finite volumes. It is usually used in conjunction with unstructured meshes and provides the best approximation property when applied to elliptic and parabolic problems at relatively low Peclet numbers, see [16]. The development of high-resolution finite element schemes for hyperbolic and convection-dominated transport equations is a topic of active research.

The finite element method is supported by a large body of mathematical theory that makes it possible to obtain rigorous error estimates and proofs of convergence. The finite element mesh (also called triangulation) is usually unstructured, and the shape of mesh cells can be fitted to the shape of a curvilinear boundary. Matrix assembly is performed element-by-element in a fully automatic way. The remarkable generality and flexibility of the FEM makes it very powerful. Almost all codes for structural mechanics problems are based on finite element approximations, and a lot of current research is aimed at the development of adaptive FEM for fluid dynamics.

Finite elements and finite volumes have a lot in common and are largely equivalent in the case of low-order polynomials. The traditional strengths of FVM and FEM, as applied to the CDR equation, are complementary. Convective terms call for the use of an upwind-biased discretization which is easier to construct within the finite volume framework. On the other hand, the finite element approach

takes the lead when it comes to the discretization of diffusive terms. Therefore, many hybrid FVM-FEM schemes have been proposed. For example, finite element shape functions are used to interpolate the fluxes for a vertex-centred finite volume method [17]. Conversely, FVM-like approximations of convective terms are frequently employed to achieve the up-winding effect in finite element codes.

A current trend in CFD is towards the use of *discontinuous Galerkin* (DG) methods [18] which represent a generalization of FVM and incorporate some of their most attractive features, such as local conservation. At the same time, the treatment of diffusive fluxes is not straightforward and requires special care, as in the case of classical FVM. We welcome the advent of DG methods but feel that the potential of continuous finite elements has not yet been exploited to the full extent in the context of transport equations.

2.6 Kratos solver

The essay aims to improve one of the Kratos strategies for solving pure convection problems as commented before. So, it is necessary to have some information about the software that is going to be improved. In this sense, this chapter contains an overview of what Kratos is and how it operates. The problems that Kratos can solve and how it solves them will be the general description of Kratos.

2.6.1 Introduction to Kratos

Kratos is a framework for building multi-disciplinary finite element programs. It provides several tools for easy implementation of finite element applications and common platform providing effortless interaction between them. Kratos has innovative variable base interface designed to be used at different levels of abstraction and implemented to be very clear and extendible. It also provides an efficient yet flexible data structure which can be used to store any type of data in a safety way. The *Python* scripting language is used to define the main procedures of Kratos which significantly improves the flexibility of the framework in time of use.

In the modelling process, Kratos is the solver that reads the information related to the geometry, boundary conditions, loads, impose displacements and mesh data. Then it transforms all this data in algebraic equations that are solved using numerical methods. Finally it outputs the solution of the numerical problem to be post-processed. The tool used for the pre-processing and post-processing steps is GiD [5].

Figure 7. Kratos in the modelling process

2.6.2 Kratos' structure

Kratos is a multi-physic tool, meaning that can model the combination of different analysis (thermal, fluid dynamic, structural) with optimised methods in the global software package with just one user interface and, even more, the possibility to extend the implemented solution to new problems.

The solver is based on the finite element method. Many problems in engineering and applied science are governed by partial differential equations, easily handled by computers thanks to numerical methods. It also uses modern object oriented philosophy from the computational point of view, due to the integration of disciplines, in the physical as well as in the mathematical sense. The modular design, hierarchy and abstraction of these approaches fits to the generality, flexibility and reusability for the current and future challenges in numerical methods. The structure of the solver is based on finite element technology and many objects are design to represent the basic finite element concepts. In this way the structure becomes easily understandable for developers with a finite element method knowledge. In this design *Vector*, *Matrix* and *Quadrature* represent the basic numerical concepts. *Node*, *Element*, *Condition* and *Dof* are defined directly from finite element concepts. *Model*, *Mesh* and *Properties* are from the practical methodology used in finite element modelling complemented by *ModelPart* and *SpatialContainer* for better organising all the data needed for the analysis. *IO*, *LinearSolver*, *Process* and *Strategy* represent the different steps of a finite element program flow. Finally *Kernel* and *Application* are defined for library management and its interface definition.

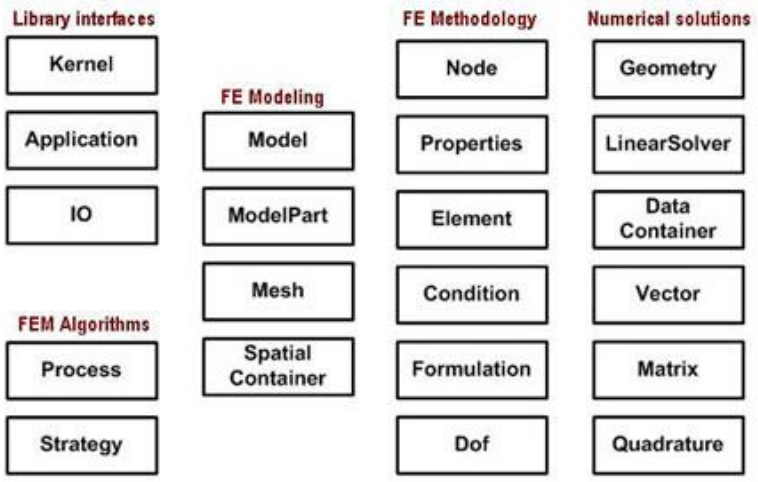


Figure 8. Kratos object-oriented structure

Kernel refers to the core, the centre of the computations. It provides the basic infrastructure end the generic numerical tools, so it is the base where the different applications rely on.

Applications consist on an adaptations of Kratos depending the field of the problem that is going to be faced. An application provides a set of specific algorithms that are needed for solving a problem from s certain field, such as fluid dynamics or solid mechanics. In the case of problems from varied fields, Kernel also allows some interaction between different applications.

The main advantage that this structure provides is that there is a clear distinction between the numerical base of the code and the parts that are related to the simulations of a particular problem. So, a lot of issues are avoided when developing applications, and the Kratos developers can focus on the code extension without worrying about the error that can be made in other parts, besides the computational time is significantly reduced.

All the tools used in Kratos are programmed in *C++* language, but they have an interface to be called from *Phyton*, that permits to the users make some modifications. Here is where the *processes* and *utilities* concepts appear. They are classes which can be used to implement the modifications in the simulations.

A *process* is a tool used to implement general tasks, but they are not covered in the conventional process of solution iteration.

Utilities are a set of tools used to do a particular task, such as mathematical functions or parallelization tools.

2.6.3 Phyton scripts

Phyton is a programming language characterised for having a simple and clear syntax. It is used for Kratos with the aim of combining in a more flexible and dynamic way the components that carry out different type of simulations. The main script of Phyton perform the role of the main function of the solver. It can be easily modified by users and developers.

The big advantage of these scripts is that changes can be quickly made. For example, the tolerance of an iterative method can be modified, as well as the boundary conditions of the problem. On the other hand, it should be taken into account that the process will be slower, so its use should be studied.

Thus, Python scripts can accomplish the following actions:

- Load Kratos and import the application
- Read problem data and create the *ModelPart* of Kratos. That is editing the mesh, elements, materials and boundary conditions.
- Define the constructor and solver
- Call the solver
- Write a file with the mesh and results

In summary, once Kratos is compiled, the code programmed with C++ is used for the application. The rest of tasks can be carried out from Python. The output script acts as an interface or process between the outline of the problem and the Kratos' functions, adding additional functions that are needed in the computation and are not used in the conventional way.

CHAPTER 3

Objectives

The main goal that is pursued in this work is to **implement a shock-capturing strategy in the BFECC method for Kratos** in order to eliminate the spurious numerical oscillations that this method produces in the solution.

With a view to achieving the commented objective, the thesis looks for secondary purposes for understanding the problem and analysing the appropriation of the strategy that is going to be implemented:

- A review of the physic context in which the problem is placed. Comment the principal mathematic concepts for solving partial differential equation, as well as the main numerical techniques that currently exist.
- An overview of Kratos tool and the Convection-Diffusion application for comprehending how this solver works, what kind of problems can it solve and what weaknesses does it have.
- A comparison analysis between the actual method implemented in Kratos and the improvement that is pretended to put into practice with several numerical examples.
- Deeply study the deficiencies of the BFECC method and when particularly it performs numerical oscillations.
- A detailed explanation of the code that is going to be implemented for the sake of clarity.
- A proper validation of the implemented shock-capturing strategy, testing several problems with different characteristics.

CHAPTER 4

Numerical algorithms and Shock-capturing strategy

In this section the schemes and algorithms that are needed to properly analyse the behaviour of the shock-capturing strategy are presented. First, the scheme of the Upwind method is presented. The Upwind method is one of the easiest methods that one can use to solve the advection equation. It approximates the derivatives of the equation following a finite-difference scheme and it is used in this work for finding the strengths of the BFECC method currently implemented in Kratos. The stability of the BFECC method strongly depends on the Courant-Friedrichs-Lewy (CFL) condition, so it is previously presented. Then, a detailed explanation of the BFECC method is then presented, as well as the scheme implemented in Kratos for describing the motion, that is the semi-Lagrangian scheme. Finally, the chapter goes into the particulars of the shock-capturing procedure.

4.1 Courant-Friedrichs-Lewy

The Courant-Friedrichs-Lewy (CFL) condition is a necessary condition for convergence while solving certain partial differential equations numerically by finite difference schemes. It arises in the numerical analysis of time integration schemes, when these are used for the numerical solution. As a consequence, the time step must be less than a certain time in many computer simulations, otherwise the simulation will produce incorrect results.

The principle behind this condition is that, for example, if a wave is moving across a discrete spatial grid and we want to compute its amplitude at discrete time steps of equal duration, then this duration must be less that the time for the wave to travel to adjacent grid points. As a corollary, when the grid point separation is reduced, the upper limit for the time step also decreases. In essence, the numerical domain of dependence of any point in space and time (as determined by initial conditions and the parameters of the approximation scheme) must include the analytical domain of dependence (wherein the initial conditions have an effect on the exact value of the solution at that point) in order to assure that the scheme can access the information required to form the solution. Further information can be found in [20].

The spatial coordinates, that are the coordinates of the physical space in which the problem is posed, and the time, as the coordinate distinct from the spatial coordinates acting as a parameter that describes the evolution of the system, are supposed to be discrete-valued independent variables, which are placed at a regular distances called the *interval length* and the *time step*. The CFL condition relates the length of the time step to a function of the interval lengths of each spatial coordinate and of the maximum speed with which information can travel in the physical space.

Essentially, the CFL condition is commonly prescribed for those terms of the finite difference approximation of general partial differential equations which model the advection phenomenon.

For the one-dimensional case, the CFL has the following form:

$$CFL = \frac{a\Delta t}{\Delta x} \leq C_{max} \tag{20}$$

where the dimensionless number is called the *Courant number*,

a is the magnitude of the velocity field;

Δt is the time step;

Δx is the length interval.

The value of C_{max} changes with the method used to solve the discretised equation, especially depending on whether the method is explicit or implicit. If an explicit integration solver is used then typically $C_{max} = 1$. Implicit solvers are usually less sensitive to numerical instability and so larger values of C_{max} may be tolerated.

In the two-dimensional case, the CFL condition becomes

$$CFL = \frac{a_x\Delta t}{\Delta x} + \frac{a_y\Delta t}{\Delta y} \leq C_{max} \tag{21}$$

By analogy with the two-dimensional case, the general CFL condition for the n-dimensional case is the following one:

$$CFL = \Delta t \sum_{i=1}^{n} \frac{a_{x_i}}{\Delta x_i} \leq C_{max} \tag{22}$$

The interval length is not required to be the same for each spatial variable, Δx_i. This characteristic can be used in order to somewhat optimize the value of the time step for a particular problem, by varying the values of the different interval in order to keep it not too small.

The CFL condition is a necessary condition, but may not be sufficient for the convergence of the finite difference approximation of a given numerical problem. Thus, in order to establish the convergence of the finite difference approximation, it is necessary to use other methods, which in turn could imply limitation on the length of the time stem and/or the lengths of spatial intervals.

4.2 The Upwind scheme

One of the simplest finite-difference schemes for the advection equation is the upwind method, which employs forward difference in time and backward difference in space considering the inverse direction of the flux. Despite the fact that it is easy to implement, two point upwind method is only first-order accurate and shows dissipation that spreads discontinuity and damps the solution. Increasing the order of the method may lead to numerical instabilities and nonphysical results.

Making use of finite-difference techniques to derive a discrete representation of the advection equation (11) by first considering the derivative in time. Taylor expanding the solution around $u(x_i, t^n)$ one obtains

$$u(x_i, t^n + \Delta t) = u(x_i, t^n) + \frac{\partial u}{\partial t}(x_i, t^n)\Delta t + \mathcal{O}(\Delta t^2) \tag{23}$$

or, equivalently,

$$u_i^{n+1} = u_i^n + \left.\frac{\partial u}{\partial t}\right|_i^n \Delta t + \mathcal{O}(\Delta t^2) \tag{24}$$

Isolating the **time derivative** and dividing by Δt it can be obtained:

$$\left.\frac{\partial u}{\partial t}\right|_i^n = \frac{u_i^{n+1} - u_i^n}{\Delta t} + \mathcal{O}(\Delta t) \tag{25}$$

Adopting a standard convention, it is usual to consider the finite-difference representation m-th order differential operator $\partial^m / \partial x^m$ in the generic x-direction (where x could either be a time or a spatial coordinate) to be of order p if and only if

$$\frac{\partial^m u}{\partial x^m} = \mathcal{L}_\Delta(u) + \mathcal{O}(\Delta x^p) \tag{26}$$

Of course, the time and spatial operators may have finite-difference representations with different orders of accuracy and in this case the overall order of the equation is determined by the finite differential operator with the largest truncation error.

Note also that while the truncation error is expressed for the differential operator, the numerical algorithms will not be expressed in terms of the differential operators and will therefore have different (usually smaller) truncation errors. This fact is clearly illustrated by the equations above, which show that the explicit solution (24) is of higher order than the finite-difference expression for the differential operator (25).

With this definition in mind, it is not difficult to realize that the finite-difference expression (25) for the time derivative is only first-order accurate in Δt. However, accuracy is not the most important requirement in numerical analysis and a first-order but stable scheme is greatly preferable to one which is higher order (i.e., has a smaller truncation error) but is unstable.

In way similar to what it has been done in (25) for the time derivative, one can derive a first-order, finite-difference approximation to the **space derivative** as

$$\left.\frac{\partial u}{\partial x}\right|_i^n = \frac{u_i^n - u_{i-1}^n}{\Delta x} + \mathcal{O}(\Delta x) \tag{27}$$

While formally similar, the approximation (27) suffers of the ambiguity, not present in expression (25), that the first-order term in the Taylor expansion can be equally expressed in terms of u_{i+1}^n and u_i^n, that is,

$$\left.\frac{\partial u}{\partial x}\right|_i^n = \frac{u_{i+1}^n - u_i^n}{\Delta x} + \mathcal{O}(\Delta x) \tag{28}$$

This ambiguity is the consequence of the first-order approximation which prevents a proper "centring" of the finite-difference stencil. However, and as long as the concern is about dealing with an advection equation, this ambiguity is easily solved if we think that the differential equation will simply translate each point in the initial solution to the new position $x + a\Delta t$ over a time interval Δt. In this case, it is natural to select the points in the solution at the time-level n that are "upwind" of the solution at the position i and at the time-level $n+1$, as these are the ones causally connected with u_i^{n+1}. Depending then on the direction in which the solution is translated, and hence on the value of the advection velocity a, two different finite-difference representations can be given of the advection equation (11) and these are

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = -a\left(\frac{u_i^n - u_{i-1}^n}{\Delta x}\right) + \mathcal{O}(\Delta t, \Delta x), \qquad if\ a > 0 \tag{29}$$

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = -a\left(\frac{u_{i+1}^n - u_i^n}{\Delta x}\right) + \mathcal{O}(\Delta t, \Delta x), \qquad if\ a < 0 \tag{30}$$

respectively. As a result, the final finite-difference algorithms for determining the solution at the new time-level will have the form

$$u_i^{n+1} = u_i^n - \frac{a\Delta t}{\Delta x}(u_i^n - u_{i-1}^n) + \mathcal{O}(\Delta t^2, \Delta x \Delta t), \qquad if\ a > 0 \tag{31}$$

$$u_i^{n+1} = u_i^n - \frac{a\Delta t}{\Delta x}(u_{i+1}^n - u_i^n) + \mathcal{O}(\Delta t^2, \Delta x \Delta t), \qquad if\ a < 0 \tag{32}$$

The discrete derivative operators defined in the forward and backward differences are exact for the linear combination of polynomial functions up to first degree. In other words, the numerical solution of the advection equation produced by upwind method is the set of discrete values of a first degree piecewise polynomial function in x and t.

More in general, for a system of linear hyperbolic equations with state vector $\boldsymbol{U}$ and flux-vector $\boldsymbol{F}$, the upwind scheme will take the form

$$\boldsymbol{U}_i^{n+1} = \boldsymbol{U}_i^n \pm \frac{\Delta t}{\Delta x}\left(\boldsymbol{F}_{i\pm1}^n - \boldsymbol{F}_i^n\right) + \mathcal{O}(\Delta t^2, \Delta x \Delta t) \tag{33}$$

where the $\pm$ sign should be chosen according to whether $a > 0$ or $a < 0$.

The logic behind the choice of the stencil in an upwind method is illustrated in Figure 9 where a schematic diagram for the two possible values of the advection velocity is shown.

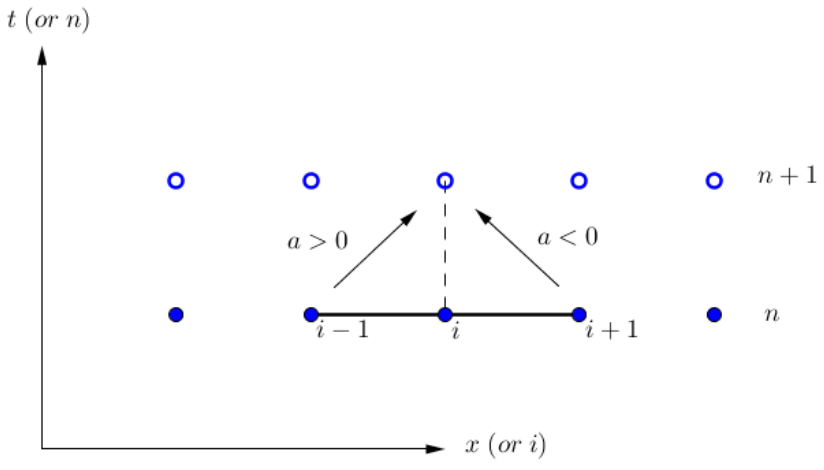


Figure 9. Triangular property distribution

The upwind scheme is an example of an explicit scheme, that is of a scheme where the solution at the new time-level $n + 1$ can be calculated explicitly from the quantities that are already known at the previous time-level n. This is to be contrasted with an implicit scheme in which the finite-difference

representations of the differential equation has, on the right-hand-side, terms at the new time-level $n+1$. These methods require in general the solution of a number of coupled algebraic equations. The upwind scheme is a stable one in the sense that the solution will not have exponentially growing modes.

4.3 Numerical approach: Semi-Lagrangian scheme

There exists two classical strategies in continuum mechanics to describe the motion. The **Lagrangian** specification of the field is a way of looking at fluid motion where the observer follows an individual fluid particle as it moves through space and time. Plotting the position on an individual parcel though time gives the pathline of the parcel. So, we observe a *material point* (a particle) and we follow it over time:

$$\boldsymbol{X} = [X, Y, Z] \tag{34}$$

We refer to *material derivative* when the scalar density function variation depends only on the scalar field:

$$d\varphi[x, y, z, t] = \frac{\partial \varphi(t)}{\partial t} dt \tag{35}$$

Rewriting the equation we obtain:

$$\frac{d\varphi}{dt} = \frac{\partial \varphi}{\partial t} \tag{36}$$

On the other hand, the **Eulerian** specification of the flow filed is a way of looking at fluid motion that focuses on specific locations in the space through which the fluid flows as time passes. So, we look at one point in space (*spatial point*) and not on the particles passing by. Many different particles are passing through the same location (*spatial* point) over time:

$$\boldsymbol{x} = [x(t), y(t), z(t)] \tag{37}$$

We refer to *local derivative* when the scalar field variation depends on the particle which, in this moment, passes through the position studied: particles move with a certain speed and at each instant of time one new particle is passing through the selected position (*spatial point*).

$$\begin{aligned} d\varphi[x(t), y(t), z(t)] &= \frac{\partial \varphi}{\partial t} dt + \frac{\partial \varphi}{\partial x} \frac{dx(t)}{dt} dt + \frac{\partial \varphi}{\partial y} \frac{dy(t)}{dt} dt + \frac{\partial \varphi}{\partial z} \frac{dz(t)}{dt} dt \\ &= \frac{\partial \varphi}{\partial t} dt + u \frac{\partial \varphi}{\partial x} dt + v \frac{\partial \varphi}{\partial y} dt + w \frac{\partial \varphi}{\partial z} dt = \left(\frac{\partial \varphi}{\partial t} + \boldsymbol{a} \cdot \nabla \varphi \right) dt \end{aligned} \tag{38}$$

Where $\boldsymbol{a}$ is the velocity field

$$\boldsymbol{a}(u, v, w) = \left(\frac{dx(t)}{dt}, \frac{dy(t)}{dt}, \frac{dz(t)}{dt} \right) \tag{39}$$

And $\boldsymbol{a} \cdot \nabla\boldsymbol{\varphi}$ is the *convective term*, which means that the scalar field is transported according to the velocity field.

The **semi-Lagrangian** scheme is a numerical method that is widely used in fluid dynamics for the integration of the equations governing the fluid motion. A Lagrangian description of a system focuses on following individual air parcels along their trajectories as opposed to the Eulerian description, which considers the range of change of system variables fixed at a particular point in space. A semi-Lagrangian scheme uses Eulerian framework but the discrete equations come from the Lagrangian perspective. In this sense, semi-Lagrangian schemes use a regular grid (Eulerian perspective), just like finite-difference methods. The idea is that at every time step the point where a parcel originated from is calculated. An interpolation scheme is then utilized to estimate the value of the dependent variable at the grid points surrounding the points where the particle originated from.

In this report we consider numerically simulating the advection of a scalar field $\boldsymbol{\varphi}$ with a given velocity vector field $\boldsymbol{a}(\boldsymbol{x}, t)$. The evolution of $\boldsymbol{\varphi}$ is governed by the advection equation:

$$\frac{\partial \varphi}{\partial t} + \boldsymbol{a} \cdot \nabla\varphi = 0 \tag{40}$$

equipped with the appropriate initial conditions.

The semi-Lagrangian method is a computational technique to solve (40). Within the semi-Lagrangian approach the approximate solution is found in time steps $t^n, n = 0, 1, 2 \ldots$ by numerical integration backward and forward in time of the characteristic equations

$$\frac{d\boldsymbol{x}(\tau)}{d\tau} = \boldsymbol{a}(\boldsymbol{x}(\tau), \tau), \quad \boldsymbol{x}(t^{n+1}) = \boldsymbol{x}_0, \quad \tau \in [t^{n+1}, t^n] \tag{41}$$

for all $\boldsymbol{x}_0$ lying in a computational domain at time t^{n+1}. If the solution at time step n is known everywhere, it is set that

$$\varphi^{n+1}(\boldsymbol{x}_0) = \varphi^n(\boldsymbol{x}(t^n)) \tag{42}$$

In practice, a spatial discretization is applied, for example defining $\boldsymbol{\varphi}$ in a finite number of nodes, which form a grid. Since the location of the solution after integrating in time $\boldsymbol{x}(t^n)$ is not necessary placed in a grid node, an interpolation φ_I^n from nodal values $\boldsymbol{\varphi}^n$ is done to define its value in $\boldsymbol{x}(t^n)$. The numerical integration of (41) and the interpolation error contribute to the numerical error of a semi-Lagrangian method. This error is not monotonic with respect to time step Δt and has the form:

$$O\left((\Delta t)^k + \frac{(\Delta x)^{p+1}}{\Delta t}\right) \tag{43}$$

where k refers to the order of the numerical integration of (41), Δx is the spatial mesh size, and p is the interpolation order of $\boldsymbol{\varphi}_I^n$.

The error estimate (43) calls for a higher order interpolation. However, standard linear higher order interpolation techniques lead to the loss of the monotonocity property, which is critical for numerical stability in many applications. Therefore, several limiting and accuracy improving techniques have been suggested in the literature allow for accurate and stable semi-Lagrangian approach.

In addition, we also need the reverse semi-Lagrangian method given by a numerical integration forward in time:

$$\frac{d\tilde{\boldsymbol{x}}(\tau)}{d\tau} = \boldsymbol{a}(\tilde{\boldsymbol{x}}(\tau), \tau), \quad \tilde{\boldsymbol{x}}(t^{n+1}) = \boldsymbol{x}_0, \quad \tau \in [t^{n+1}, t^n] \tag{44}$$

and setting $\tilde{\boldsymbol{\varphi}}^n(\boldsymbol{x}_0) = \tilde{\varphi}_l^{n+1}(\tilde{\boldsymbol{x}}(t^{n+1}))$. Here $\tilde{\varphi}_l^{n+1}$ denotes a suitable interpolation of $\tilde{\boldsymbol{\varphi}}^{n+1}$ at $\tilde{\boldsymbol{x}}(t^{n+1})$.

4.4 Back and Forth Error Compensation and Correction

Back and Forth Error Compensation and Correction (BFECC) method is a numerical technique to improve the accuracy of a semi-Lagrangian without evoking higher order interpolation. This predictor-corrector type method is based on the observation that if we solve the equation (40) forward in time for one time step using a numerical integrator and then backward in time for one time step with the same method, the difference between the two copies of the solution gives us information about the numerical error which can be used to improve the accuracy. When applied to the semi-Lagrangian convection it is observed to improve the convergence rate by one order in space and time. For a given discrete solution $\boldsymbol{\varphi}^n$ at time t^n, the semi-Lagrangian BFECC method finds $\boldsymbol{\varphi}^{n+1}$ in several steps:

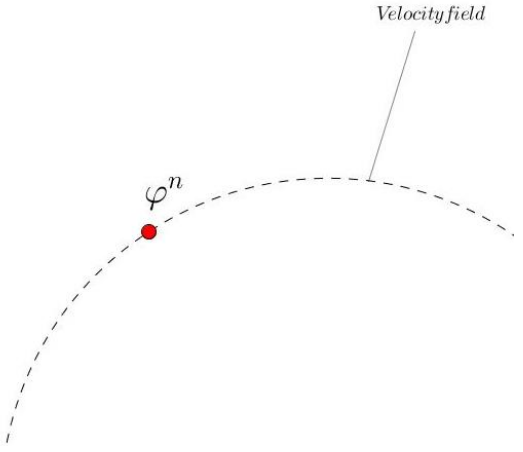


Figure 10. Initial position of $\boldsymbol{\varphi}$ at time step t^n

4.4.1 Forward advection

The first step of the method consists on solving (40) forward in time with the semi-Lagrangian method (41) to obtain the solution at time step t^{n+1}:

$$\hat{\varphi}^{n+1} = \mathcal{F}(\varphi^n) \tag{45}$$

where $\mathcal{F}$ is the numerical scheme that updates the numerical solution integrating and interpolating from the time t^n to t^{n+1}. In this step we approximate the position of every grid node convecting the particles with the actual velocity field.

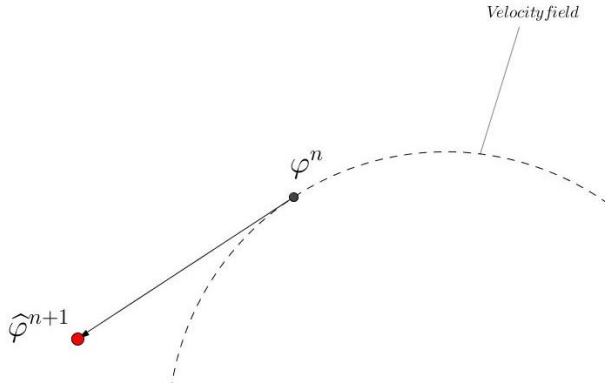


Figure 11. Position of the particle at time step t^{n+1} after moving it forwards

4.4.2 Backward advection

The second step solves (40) backward in time with the same semi-Lagrangian method to obtain $\hat{\varphi}^n$ by integrating and interpolating using the numerical scheme $\mathcal{B}$, which is the application of $\mathcal{F}$ to the time-reversed equation (44)

$$\hat{\varphi}^n = \mathcal{B}(\hat{\varphi}^{n+1}) \tag{46}$$

In this step we compute the solution of the previously moved particles once we move them back forward. If this method had no error, this operation would return the particles back to its initial position, but since the numerical solution is not exactly the right solution. An error will be computed in order to evaluate the difference between the initial solution and the solution after applying the back and forth steps.

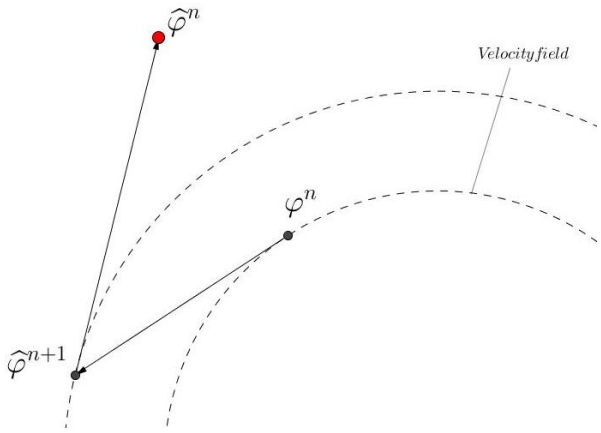


Figure 12. Position of the particle at time step t^n after moving it backwards

4.4.3 Error Compensation and Correction

In the final step of the BFECC method we compute the error made by the back and forth steps of the solution comparing the position obtained $\hat{\varphi}^n$ with the initial one φ^n as:

$$e^{(1)} = \frac{1}{2}(\varphi^n - \hat{\varphi}^n) \tag{47}$$

The difference between the initial position of the particle and the position after moving it forward and the backward is twice the error.

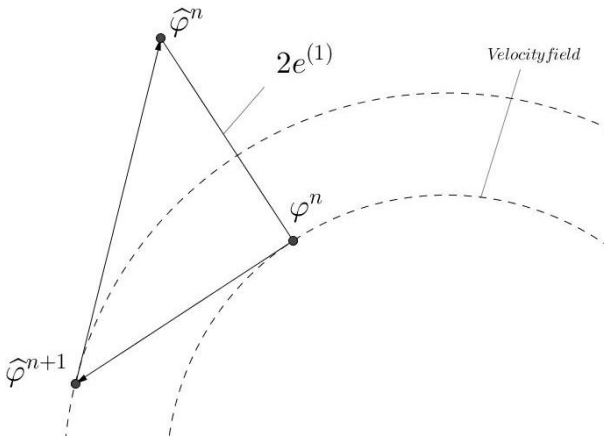


Figure 13. Error between the initial and the moved position

This error is then used to modify the initial position φ^n such that applying a new forward advection the position φ^{n+1} will not have the systematic error of the semi-Lagrangian approximation. So, the modified initial solution is:

$$\tilde{\varphi}^n = \varphi^n + e^{(1)} \tag{48}$$

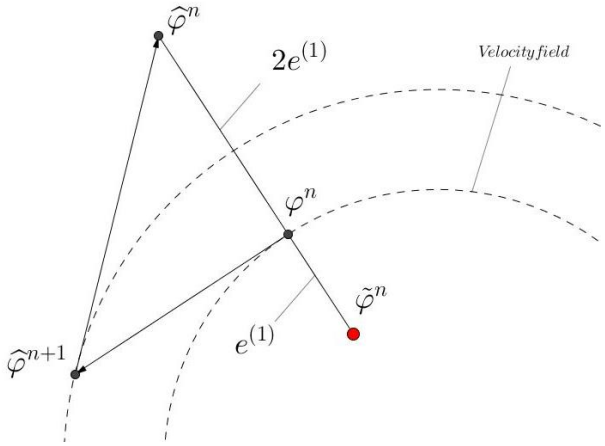


Figure 14. Correction of the particle's position at time step t^n

Finally, we solve (40) forward in time with the same semi-Lagrangian method to obtain the solution at time t^{n+1}:

$$\varphi^{n+1} = \mathcal{F}(\tilde{\varphi}^n) \tag{49}$$

The same velocity vector is used to move the corrected position particle forward in time.

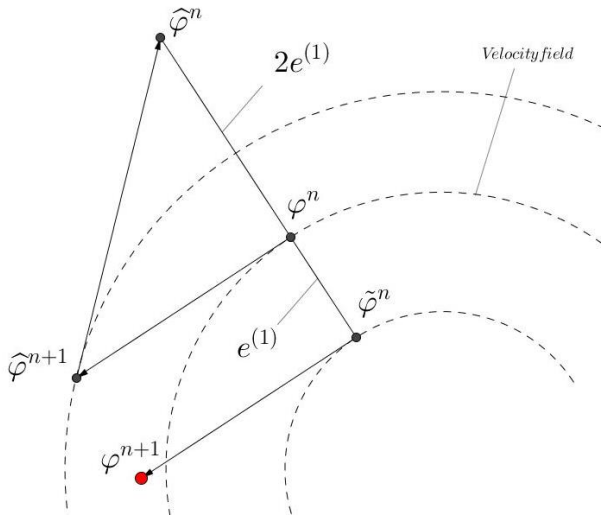


Figure 15. Particle's final position at t^{n+1} after moving the corrected position forward

4.5 Numerical oscillations in sharp fronts

The above predictor-corrector scheme is expected to improve the order of the solver subject to the interpolation accuracy and the numerical integration strategy along the characteristics. Even though, when the solution is not smooth the method is still prone to produce spurious oscillations.

If the semi-Lagrangian operators $\mathcal{F}$ and $\mathcal{B}$ are linear we can rewrite the BFECC algorithm as:

$$\varphi^{n+1} = \mathcal{F}_L\left(\frac{3}{2}\varphi^n - \mathcal{B}_L\big(\mathcal{F}_L(\varphi^n)\big)\right) \tag{50}$$

Then, the error $e^{(1)}$ holds

$$e^{(1)} = \frac{1}{2}\Big(\varphi^n - \mathcal{B}_L\big(\mathcal{F}_L(\varphi^n)\big)\Big) \tag{51}$$

Finally another error can be defined:

$$\begin{aligned} e^{(2)} &= \varphi^n - \mathcal{B}_L(\varphi^{n+1}) - e^{(1)} \\ &= \varphi^n - \mathcal{B}_L\Big(\mathcal{F}_L\big(\varphi^n + e^{(1)}\big)\Big) - e^{(1)} \\ &= \varphi^n - \mathcal{B}_L\big(\mathcal{F}_L(\varphi^n)\big) - e^{(1)} - \mathcal{B}_L\Big(\mathcal{F}_L\big(e^{(1)}\big)\Big) \\ &= 2e^{(1)} - e^{(1)} - \mathcal{B}_L\Big(\mathcal{F}_L\big(e^{(1)}\big)\Big) \\ &= e^{(1)} - \mathcal{B}_L\Big(\mathcal{F}_L\big(e^{(1)}\big)\Big) \end{aligned} \tag{52}$$

In non-smooth areas, $\left|e^{(2)}\right|$ could be larger than $\left|e^{(1)}\right|$. In fact at a grid point where $e^{(2)}$ is greater than $e^{(1)}$, there could be overshoots of the numerical solution caused by large values of $e^{(1)}$ at adjacent points. See proof in [21]. So, the violation of $\left|e_i^{(2)}\right| \leq \left|e_i^{(1)}\right|$ for a node i indicate the appearance of oscillations in adjacent points due to the correction and it is necessary to limit $e^{(1)}$ in such nodes.

4.6 BFECC with Shock-capturing strategy

The shock-capturing strategy implemented in BFECC method aims to first identify those nodes in which numerical oscillations occur and then modify the BFECC error for the final forward advection. Firstly, we rename the solution of the BFECC method as θ^{n+1}, so that φ^{n+1} would be the solution of the BFECC method with the implementation of the shock-capturing strategy.

$$\theta^{n+1} = \varphi^{n+1}$$

4.6.1 Backward advection to define comparative error

The comparative error $e^{(2)}$ is needed to identify those nodes in which numerical oscillations are arise. As shown in (52) the comparative error is computed with the original position, the backward advection of the BFECC solution and the BFECC error $e^{(1)}$:

$$e^{(2)} = \varphi^n - \big(\mathcal{B}(\theta^{n+1}) + e^{(1)}\big) \tag{53}$$

This error evaluates the difference between the original particle's position with the backward corrected advection of the BFECC solution. This difference should be smaller than the $e^{(1)}$ correction used in the BFECC method.

4.6.2 Limiting

The violation of $\left|e_i^{(2)}\right| \leq \left|e_i^{(1)}\right|$ for a node i indicates the appearance of oscillations as commented above, so a shock-capturing strategy of these oscillations needs to be used. The shock-capturing strategy consists on limiting the error $e^{(1)}$ in all nodes involved in the interpolation procedure for the node i. If we define j the adjacent nodes of i, the shock-capturing strategy is the following:

$$\begin{aligned} &for\ all\ nodes\ initialize && \tilde{e}^{(1)} = e^{(1)} \\ &\quad for\ every\ grid\ point\ i\ s.t. && \left|e_i^{(2)}\right| > \left|e_i^{(1)}\right| \\ &\quad\quad for\ every\ grid\ point\ j\ adjacent\ to\ grid\ point\ i && \tilde{e}_j^{(1)} = minmod\left(e_i^{(1)}, \tilde{e}_j^{(1)}\right) \end{aligned} \tag{54}$$

The BFECC error corrector $e^{(1)}$ is modified in the j adjacent nodes of i, but not in the node i.

The *minmod* function evaluates the minimum absolute value of two scalars with the same sign:

$$minmod(a,b) = \begin{cases} min(a,b), & if\ a,b > 0 \\ max(a,b), & if\ a,b < 0 \\ 0, & otherwise \end{cases} \tag{55}$$

The correction $\tilde{e}^{(1)}$ of the BFECC error $e^{(1)}$ affects only in the nodes where numerical oscillation take place.

4.6.3 Forward with modified solution

Finally, we apply the last forward advection step of the BFECC method. But, this time the original particle's position is corrected with a new error which is different from the BFECC error in such nodes where we observe numerical oscillations

$$\varphi^{n+1} = \mathcal{F}\left(\varphi^n + \tilde{e}^{(1)}\right) \tag{56}$$

CHAPTER 5

Comparison analysis of resolution methods

Kratos is currently applying the BFECC method to solve convection problems. As commented in previous chapters, this method produce numerical oscillations when the solution is far from being smooth. In order to understand the drawbacks and the shortcomings of this method and in which situations the BFECC method cause more problems, this chapter contains the performance of many numerical examples. The numerical examples have been designed to test the BFECC method in many situations, so the following parameters will be changed in every example due to their influence on the stability of the final solution:

- Initial condition. Many initial condition functions are implemented in order to test when the numerical oscillations occur. The initial condition functions will range from very smooth functions to non-smooth functions with sharp fronts. The advection of a Gaussian pulse is tested to observe the behaviour of the method while dealing with smooth functions and a square wave will be used to test the behaviour of the method in front of a non-smooth function with sharp fronts. A pyramid shape is used to solve the advection equation from an intermediate case.
- Courant number. The Courant-Friedrichs-Lewy (CFL) condition relates the length of the time step to a function of the interval lengths of each spatial coordinate and of the maximum

speed with which information can travel in the physical space, see [11]. For one-dimensional case, the CFL has the following form:

$$CFL = \frac{a\Delta t}{\Delta x} \tag{57}$$

Where a is the is the magnitude of the velocity, Δt is the time step and Δx is the length interval. The CFL condition is a necessary condition for convergence while solving partial differential equations (usually hyperbolic PDEs) numerically by the finite difference method and affects directly to the numerical solution of the problem. Further information can be found in 4.1.

The performed numerical examples are solved using the BFECC method as well as using the Upwind method described in 4.2 in order to identify when the BFECC method is better than the simplest one and when it produces too many numerical oscillations.

Kratos solves two-dimensional and three-dimensional problems. For the sake of better understanding the behaviour of the BFECC method, a Matlab code has been implemented to previously solve the one-dimensional advection equation. For the two-dimensional numerical examples, a Matlab code has been also implemented to solve problems with different initial conditions. Finally, a two-dimensional test example is solved using Kratos.

The output of the numerical examples' solution are the overshooting and undershooting values. These magnitudes inform us about the capacity of the different methods to maintain the original shape of the function that is advected. The overshooting and undershooting rates are the difference in percentage of the original and final shape of the function on the top and bottom edges.

The boundary conditions imposed to solve the numerical examples in 1D and in 2D are *periodic boundary conditions* (PBCs). PBCs are a set of boundary conditions that permit a particle passing through one side of the domain boundary re-appear on the opposite side with the same velocity.

To illustrate the periodic boundary conditions the concept of *ghost points* is introduced. The ghost points are imaginary points placed out of the physical mesh used to impose the periodic boundary conditions. In the following case, there is a linear mesh of *N-1* points. So the ghost point in the left edge of the mesh will be the *i=-1* and the ghost point in the right edge of the mesh will be *i=N*.

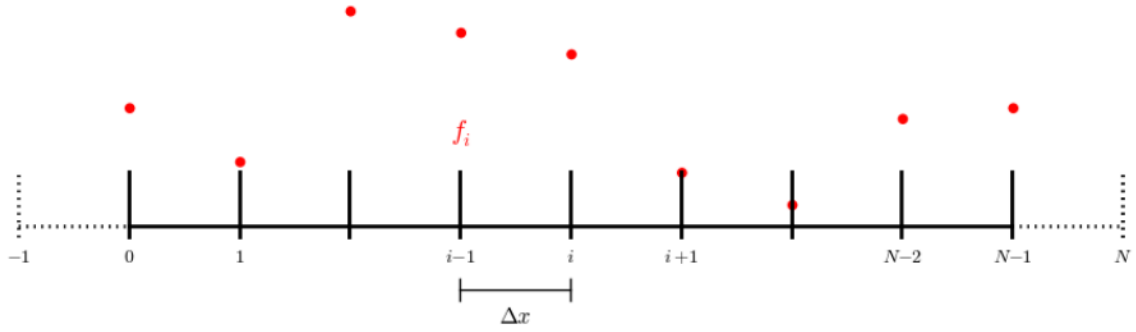


Figure 16. Periodic boundary conditions scheme

In order to impose the periodic boundary conditions, it is necessary to impose the same value in both edges with respect to the ghost point placed on the other edge for every time step n, such that:

$$f_N^n = f_1^n$$
$$f_{-1}^n = f_{N-2}^n \tag{58}$$

5.1 Numerical solution for linear advection equation in 1D

The 1D numerical examples consists on solving the following linear advection:

$$\frac{\partial u(x,t)}{\partial t} + a\frac{\partial u(x,t)}{\partial x} = 0 \tag{59}$$

with periodic boundary conditions:

$$u(0,t) = u(1,t) \tag{60}$$

in a [0,1] domain subjected to several initial conditions $u(x,0)$. As commented before, the shape of the initial condition function directly influences on the behaviour of the numerical method used to solve the advection equation. To test that, the advection of a square wave, a pyramid wave and a Gaussian pulse will be computed with the Upwind method (see 4.2.) and the BFECC method currently implemented in Kratos varying several parameters.

The computed solution corresponds to the shape of the initial condition function after one loop in the domain.

5.1.1 1D Gaussian pulse

The Gaussian pulse simulates a smooth function which a priori do not have to cause numerical oscillations. The function that describes a Gaussian pulse in the [0,1] domain is the normal distributed probability function:

$$u(x,0|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^3}} \tag{61}$$

Where μ is the mean of the normal distribution and σ is the standard deviation. For the current numerical example it has been considered a distribution with $\mu = 0.5$ and $\sigma = 0.1$.

Considering a CFL value of 0.5, the solution of the Upwind method looks extremely diffused while the BFECC solution preserve the original shape with no numerical oscillations.

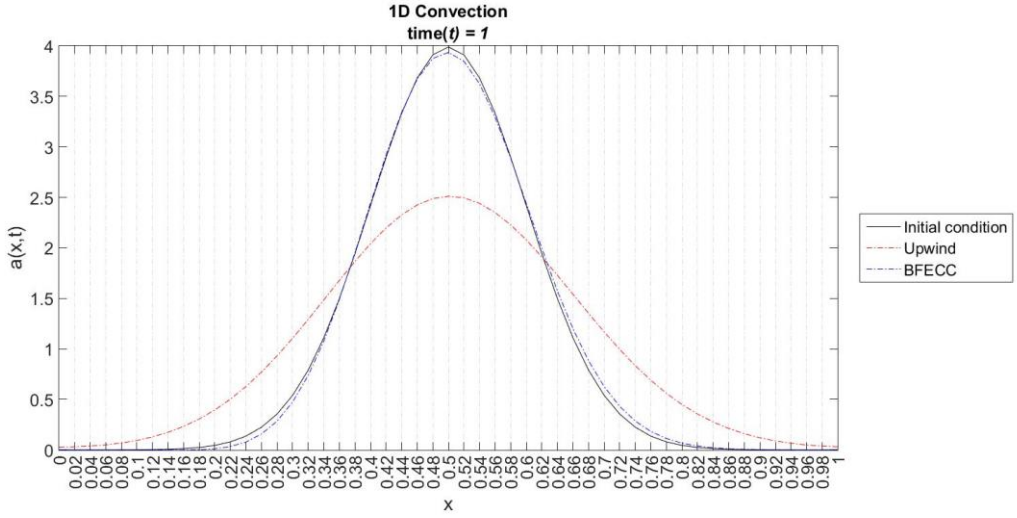


Figure 17. 1D Gaussian pulse advection with CFL = 0.5

CFL	Method	Overshooting	Undershooting
0.5	Upwind	-37.205%	0.733%
	BFECC	-1.395%	-0.044%

Table 1. 1D Gaussian pulse numerical oscillations with CFL = 0.5

Moving to a problem with a greater CFL number, up to 1.0, implies that the solution of the Upwind method is still very defused, but less than the previous case. The solutions of the BFECC still preserves the initial shape properly, although a little bit of undershooting in the top edge is observed.

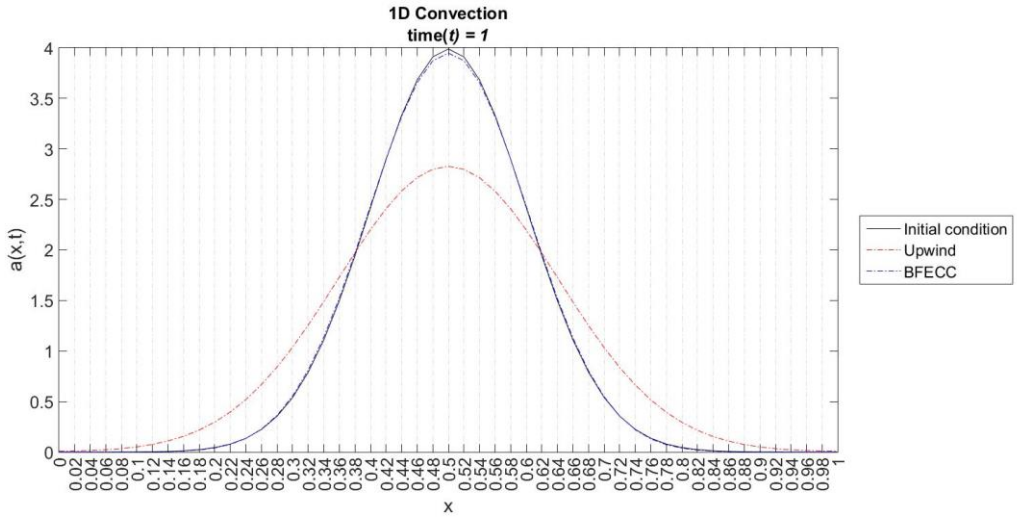


Figure 18. 1D Gaussian pulse advection with CFL = 1.0

CFL	Method	Overshooting	Undershooting
1.0	Upwind	-29.489%	0.207%
	BFECC	-1.490%	-0.001%

Table 2. 1D Gaussian pulse numerical oscillations with CFL = 1.0

Dealing with the 1D problem with CFL = 2.0 signify that the behaviour of the Upwind method is much better than the previous tests. In this case the Upwind solution is no longer diffusive. The BFECC solution preserves even better the shape of the initial function.

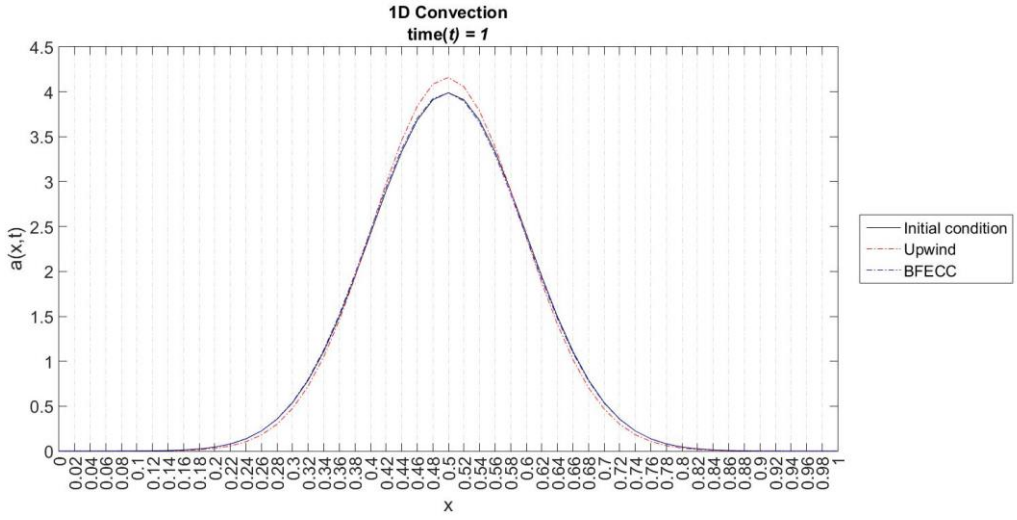


Figure 19. 1D Gaussian pulse advection with CFL = 2.0

CFL	Method	Overshooting	Undershooting
2.0	Upwind	4.367%	-0.000%
	BFECC	-0.021%	-0.000%

Table 3. 1D Gaussian pulse numerical oscillations with CFL = 2.0

5.1.2 1D Pyramid

The Pyramid function simulates a function with an abrupt change of slope, so the function is not such smooth as the Gaussian pulse. The function that describes the Pyramid in the [0,1] domain is the following:

$$u(x,0) = 1 - |2x - 1| \tag{62}$$

In the case of CFL = 0.5, the Upwind solution appears to be very diffused in both top and bottom edges. The BFECC solutions does not show numerical oscillations but some diffusion is observed in both top and bottom edges.

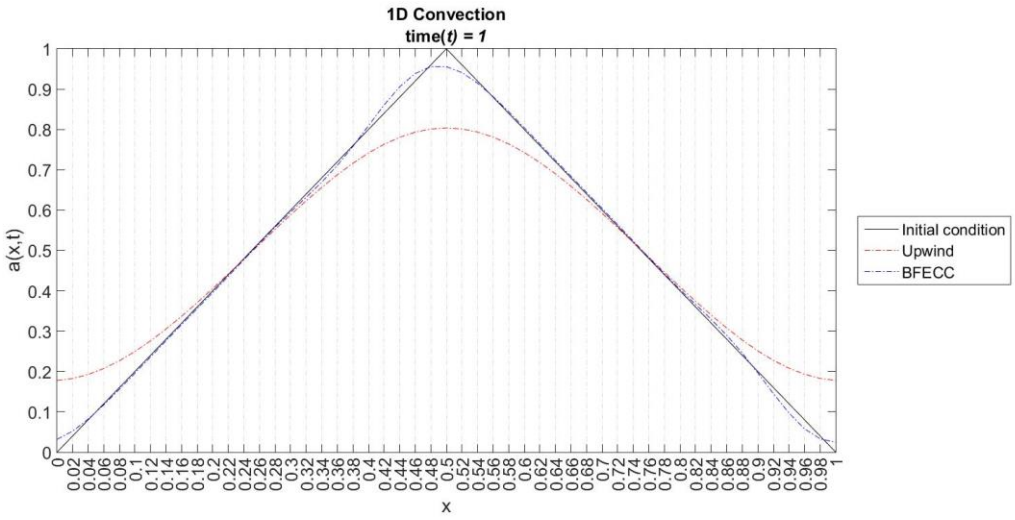


Figure 20. 1D Pyramid advection with CFL = 0.5

CFL	Method	Overshooting	Undershooting
0.5	Upwind	-19.698%	17.758%
	BFECC	-4.290%	2.504%

Table 4. 1D Pyramid numerical oscillations with CFL = 0.5

Moving to CFL = 1.0 we obtain more or less the same behaviour. The Upwind solution is still very diffused and the BFECC solution also shows diffusion in both edges. It is true that the diffusion effect is lower than with CFL = 0.5, as was also proved in the Gaussian pulse case.

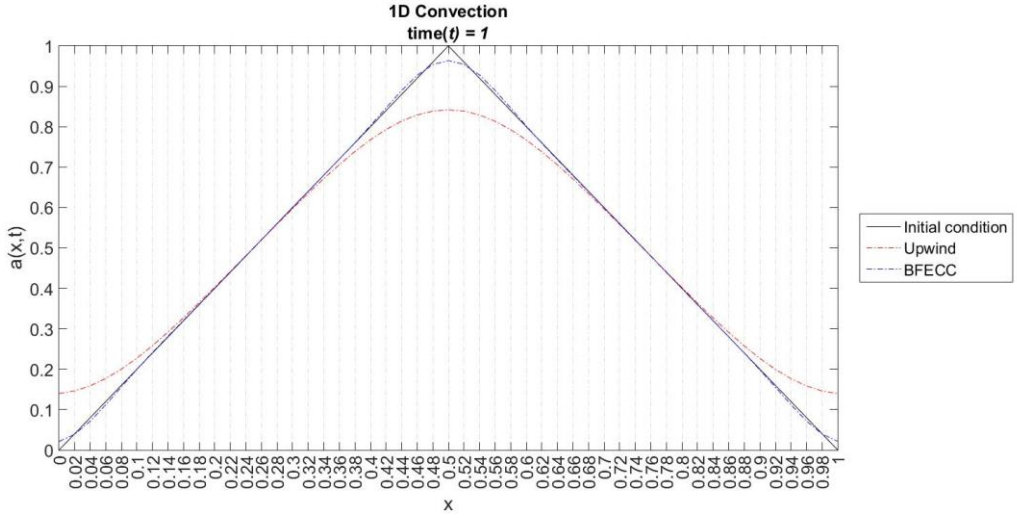


Figure 21. 1D Pyramid advection with CFL = 1.0

CFL	Method	Overshooting	Undershooting
1.0	Upwind	-15.989%	13.990%
	BFECC	-3.876%	1.905%

Table 5. 1D Pyramid numerical oscillations with CFL = 1.0

Finally, the case with CFL = 2.0 seems to be more favourable for moth methods. The Upwind solution is still diffused but much less than the previous cases. The BFECC solution preserves quite well the initial shape with no numerical oscillations but seems to diffuse a little bit in the change of slope zones.

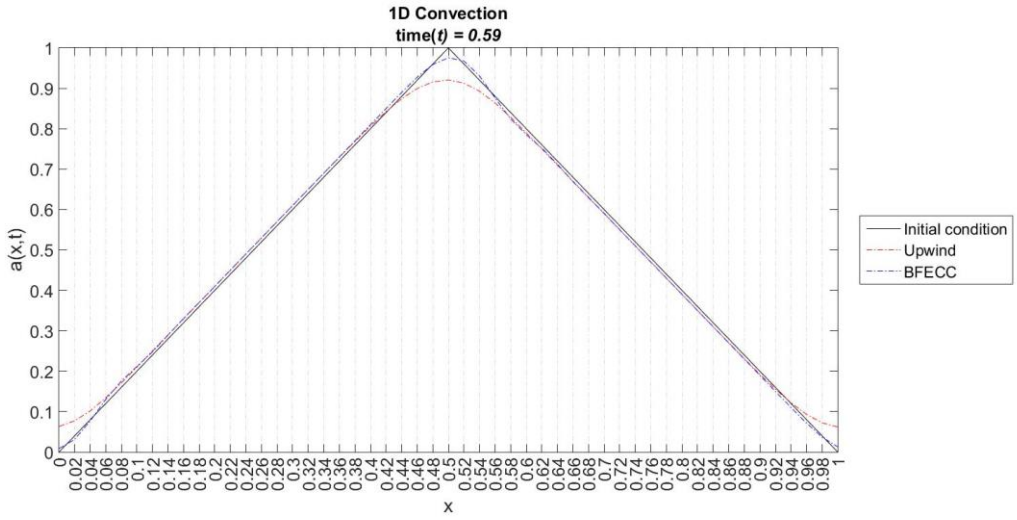


Figure 22. 1D Pyramid advection with CFL = 2.0

CFL	Method	Overshooting	Undershooting
2.0	Upwind	-8.043%	6.200%
	BFECC	-2.405%	0.991%

Table 6. 1D Pyramid numerical oscillations with CFL = 2.0

5.1.3 1D Square wave

The numerical test with a Square wave as the initial condition aims to analyse the behaviour of both methods when dealing with a non-smooth function with extreme sharp fronts. The function that defines a square wave in 1D in the [0,1] domain is the following:

$$\begin{aligned} u(x,0) &= 1, \qquad x \in \left(\frac{1}{4}, \frac{3}{4}\right) \\ u(x,0) &= 0, \qquad otherwise \end{aligned} \tag{63}$$

Starting with the case of CFL = 0.5, the diffusion is present in the Upwind solution, that cannot preserve the non-smooth initial shape but it does not perform any numerical oscillations. Conversely, in the BFECC solution one can observe the numerical oscillations that were expected to appear. On the top edge the numerical oscillations lead to a overshoot of 14.325% and on the bottom edge the undershooting is symmetric.

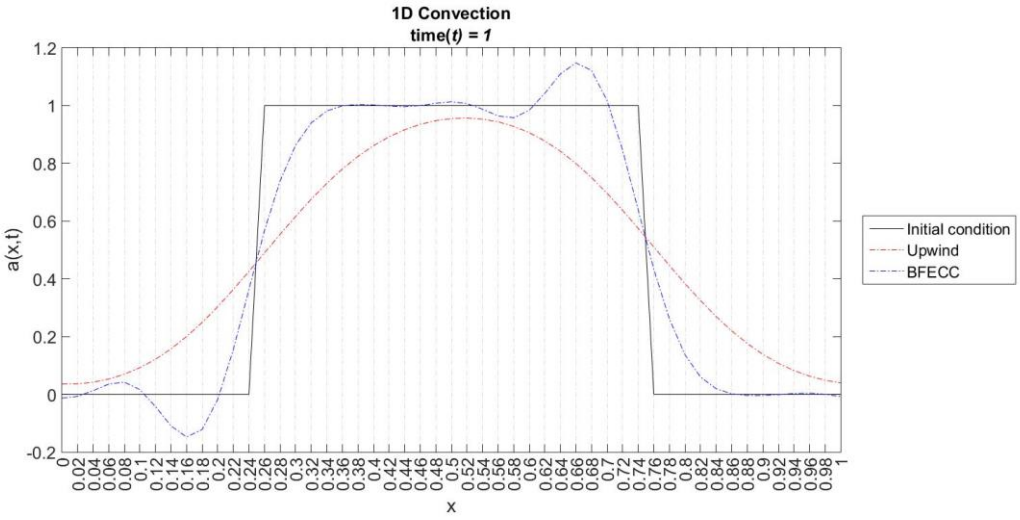


Figure 23. 1D Square wave advection with CFL = 0.5

CFL	**Method**	**Overshooting**	**Undershooting**
0.5	Upwind	-4.347%	3.605%
	BFECC	14.325%	-14.324%

Table 7. 1D Square wave numerical oscillations with CFL = 0.5

In the case of CFL = 1.0, it can be observed that the Upwind solution preserves the initial condition better than with CFL = 0.5. The Upwind solution does not show numerical oscillations but is not capable of reproducing the sharp fronts properly. The BFECC solution reproduces much better the sharp fronts than the Upwind solution, but has some numerical oscillations that lead to an overshooting and undershooting of 5.671% in both edges. The numerical oscillations are lower than in the case of CFL = 0.5.

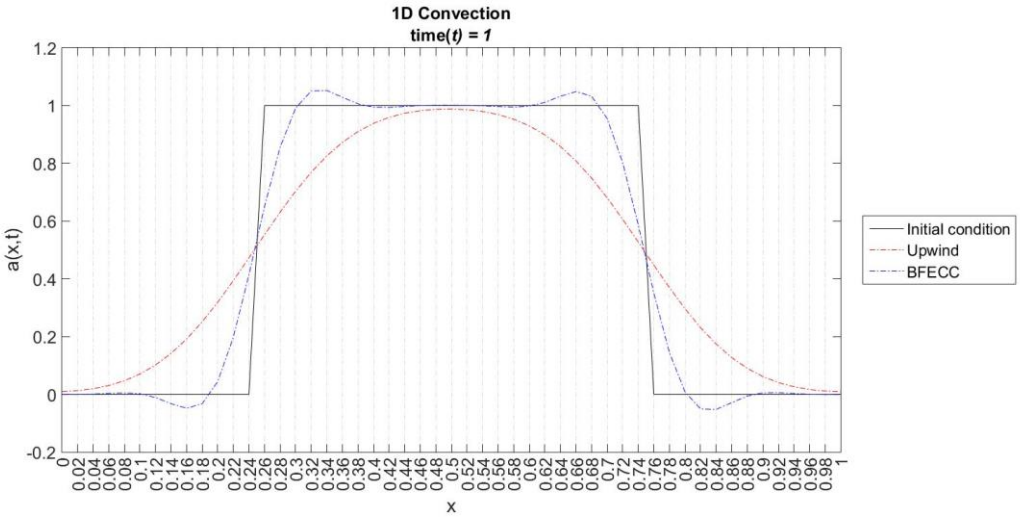


Figure 24. 1D Square wave advection with CFL = 1.0

CFL	Method	Overshooting	Undershooting
1.0	Upwind	-1.220%	0.899%
	BFECC	5.671%	-5.671%

Table 8. 1D Square wave numerical oscillations with CFL = 1.0

Finally, the solution with CFL = 2.0 performed by the Upwind method appear to be quite good, since is able to maintain both the top and the bottom edges. The Upwind solution is still not reproducing properly the sharp fronts, even though the solution is improving with the CFL number. The solution of the BFECC method leads to large numerical oscillations, that produce an overshooting and undershooting of 19.155%.

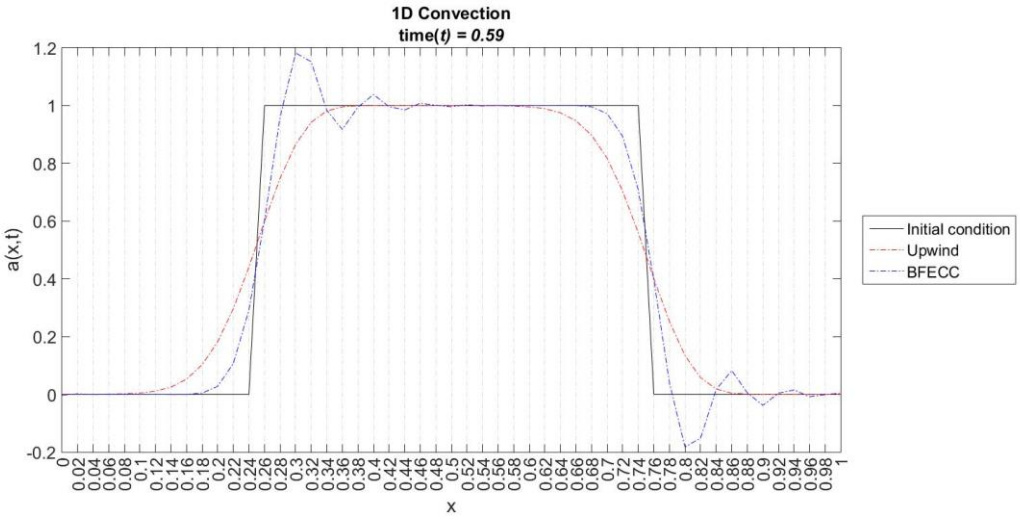


Figure 25. 1D Square wave advection with CFL = 2.0

CFL	Method	Overshooting	Undershooting
2.0	Upwind	-0.000%	0.000%
	BFECC	19.155%	-19.155%

Table 9. 1D Square wave numerical oscillations with CFL = 2.0

5.2 Numerical solution for advection equation in 2D

The 2D numerical examples consists on solving the following linear advection:

$$\frac{\partial u(\boldsymbol{x},t)}{\partial t} + a_x \frac{\partial u(\boldsymbol{x},t)}{\partial x} + a_y \frac{\partial u(\boldsymbol{x},t)}{\partial y} = 0 \tag{64}$$

with periodic boundary conditions:

$$u(x = 0, y, t) = u(x = 2, y, t) \tag{65}$$

$$u(x, y = 0, t) = u(x, y = 2, t) \tag{66}$$

in a [0,2]x[0,2] domain subjected to several initial conditions $\boldsymbol{u(x,0)}$. As in the 1D test examples, the shape of the initial condition function directly influences on the behaviour of the numerical method used to solve the advection equation. To test that influence, the advection of a square wave and a Gaussian pulse will be computed with the Upwind method (see 4.2.) and the BFECC method, currently implemented in Kratos.

The computed solution corresponds to the shape of the initial condition function after one loop in the domain.

For the 2D numerical tests, the methods have been testes under CFL = 0.75, that is, considering a velocity field in each direction of 1.5, a time step of 0.01 and a grid spacing of 0.04.

5.2.1 2D Gaussian pulse

The Gaussian pulse simulates a smooth function which a priori do not have to cause numerical oscillations. The function that describes a Gaussian pulse in a 2D domain is the following:

$$u(x,y) = Aexp\left(-\left(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2}\right)\right) \tag{67}$$

Where A is the amplitude of the pulse, (x_0,y_0) is the centre and σ_x , σ_y are the x and y spreads of the blob. For the current numerical example it has been considered a function whose centre is placed in $(x_0 = 1,y_0 = 1)$, the amplitude is $A = 0.1$ and the x and y spreads are $\sigma_x = 1, \sigma_y = 1$.

In the solution of the 2D Gaussian pulse using the Upwind method an excessive diffusion can be observed. Specifically, the peak of the function seems to diffuse around 37.23%.

CFL x direction	**CFL y direction**	**Method**	**Overshooting**	**Undershooting**
0.75	0.75	Upwind	-37.230 %	-0.178%

Table 10. 2D Gaussian pulse advected by Upwind method

In effect, the diffusion can be clearly observed from the perspective view of the function. The peak in the initial condition takes a value of 1, while in the Upwind solution the peak is around 0.6.

The symmetric shape of the initial condition is also not conserved. The function tends to deform, enlarging itself in the perpendicular direction of the movement.

In the bottom edge, it can be observed that the function also tends to raise a little bit.

Despite the diffusion effect that suffers the Upwind method, no numerical oscillations are observed in its solution after moving the initial function one loop over the domain.

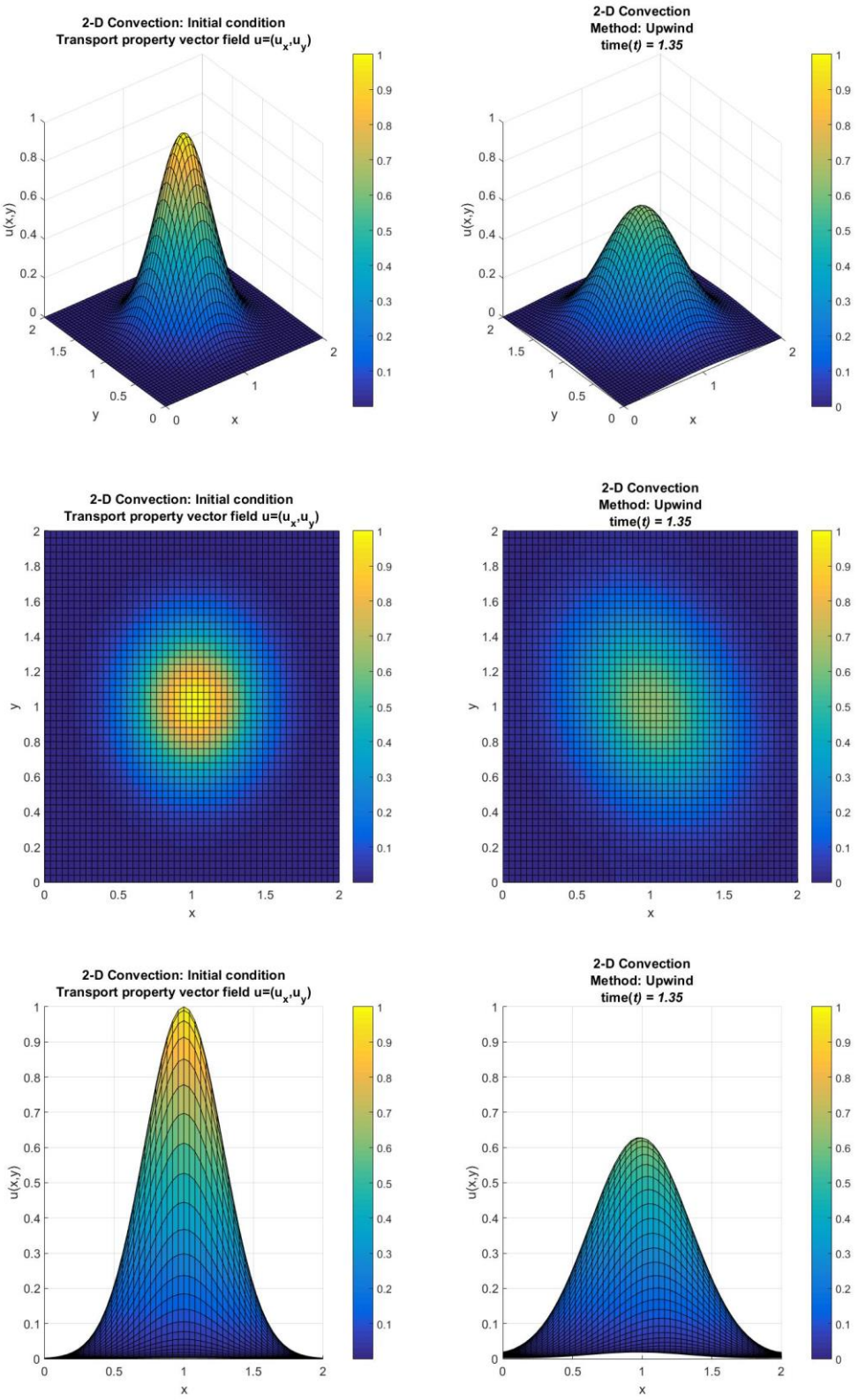


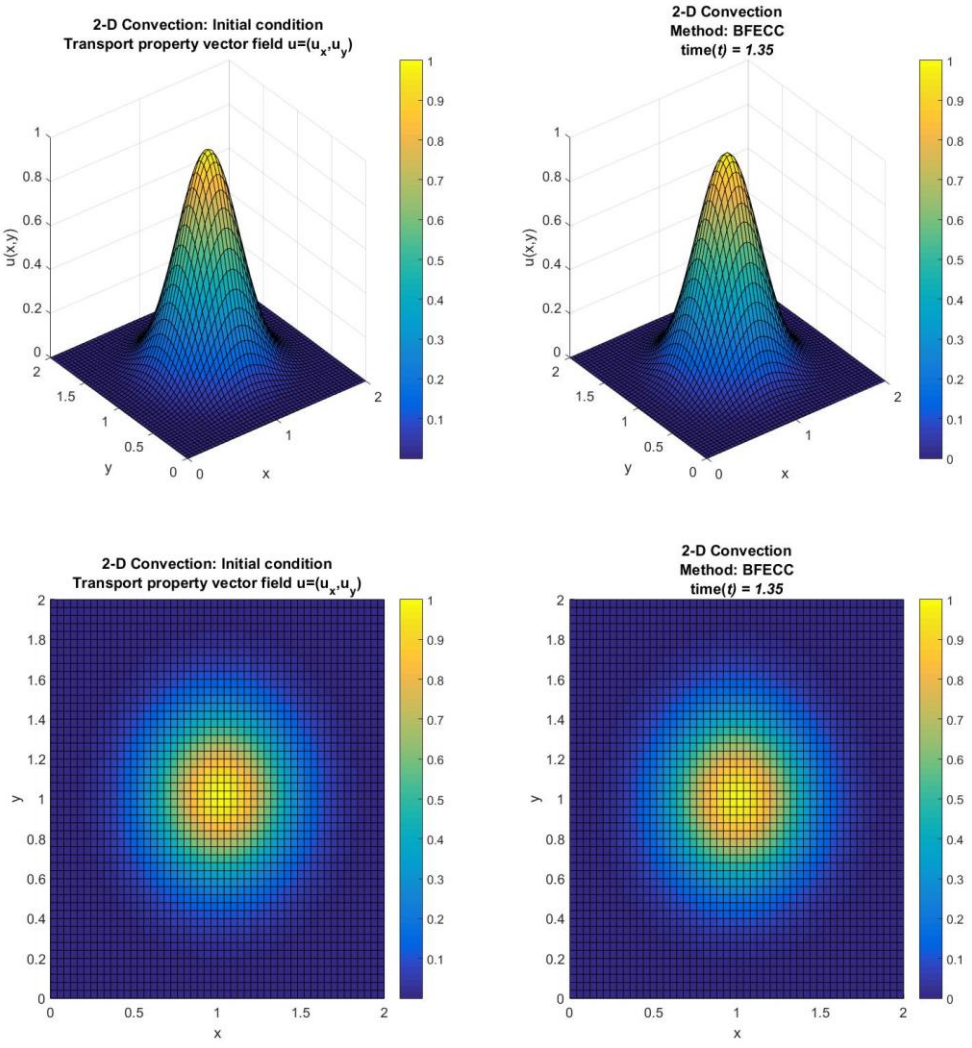
Figure 26. 2D Gaussian pulse advected by Upwind method

The following numerical test is the solution of the 2D Gaussian pulse by the BFECC method. Foremost, the BFECC method seems to preserve the initial shape of the function with no numerical oscillations neither on the bottom edge nor on the top edge.

CFL x direction	CFL y direction	Method	Overshooting	Undershooting
0.75	0.75	BFECC	-1.016 %	0.001%

Table 11. 2D Gaussian pulse advected by BFECC method

Looking at the figures, it can be observed that no numerical oscillations appear when solving a smooth function with the BFECC method. After one loop over the domain, the functions preserves almost the peak and the symmetry.



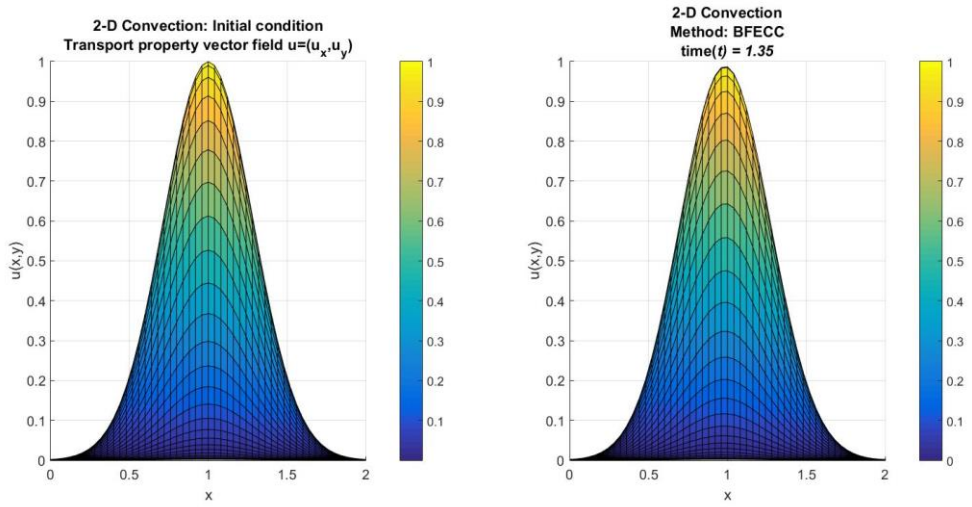


Figure 27. 2D Gaussian pulse advected by BFECC method

5.2.2 2D Square wave

The square function simulates a non-smooth function with steep sharp fronts. This function may cause problems in terms of numerical oscillations due to the non-smooth property. The square wave in function takes a value of 1 in the centre and 0 in the outer region of the 2D domain:

$$\begin{aligned} u(\boldsymbol{x},0) &= 1, \qquad u \in \left(\left[\frac{3}{4},\frac{5}{4}\right] \mathrm{X} \left[\frac{3}{4},\frac{5}{4}\right]\right) \\ u(\boldsymbol{x},0) &= 0, \qquad otherwise \end{aligned} \tag{68}$$

The Upwind method behaves in a similar way solving the 2D square wave as solving the Gaussian pulse. A lot of diffusion can be observed in the final solution. The Upwind method is not able to preserve the initial shape at all. Specifically, the square in the centre of the function seems to diffuse around 39.257%.

CFL x direction	**CFL y direction**	**Method**	**Overshooting**	**Undershooting**
0.75	0.75	Upwind	-39.257 %	0.000%

Table 12. 2D Square wave advected by Upwind method

The figures also show that the Upwind method does not produce any numerical oscillations even for very non-smooth functions. The symmetry of the function is also lost, since the function seems to enlarge in the perpendicular direction of the velocity. The diffusion of the Upwind solution leads to a Gaussian pulse shape instead of maintaining the original square shaped function.

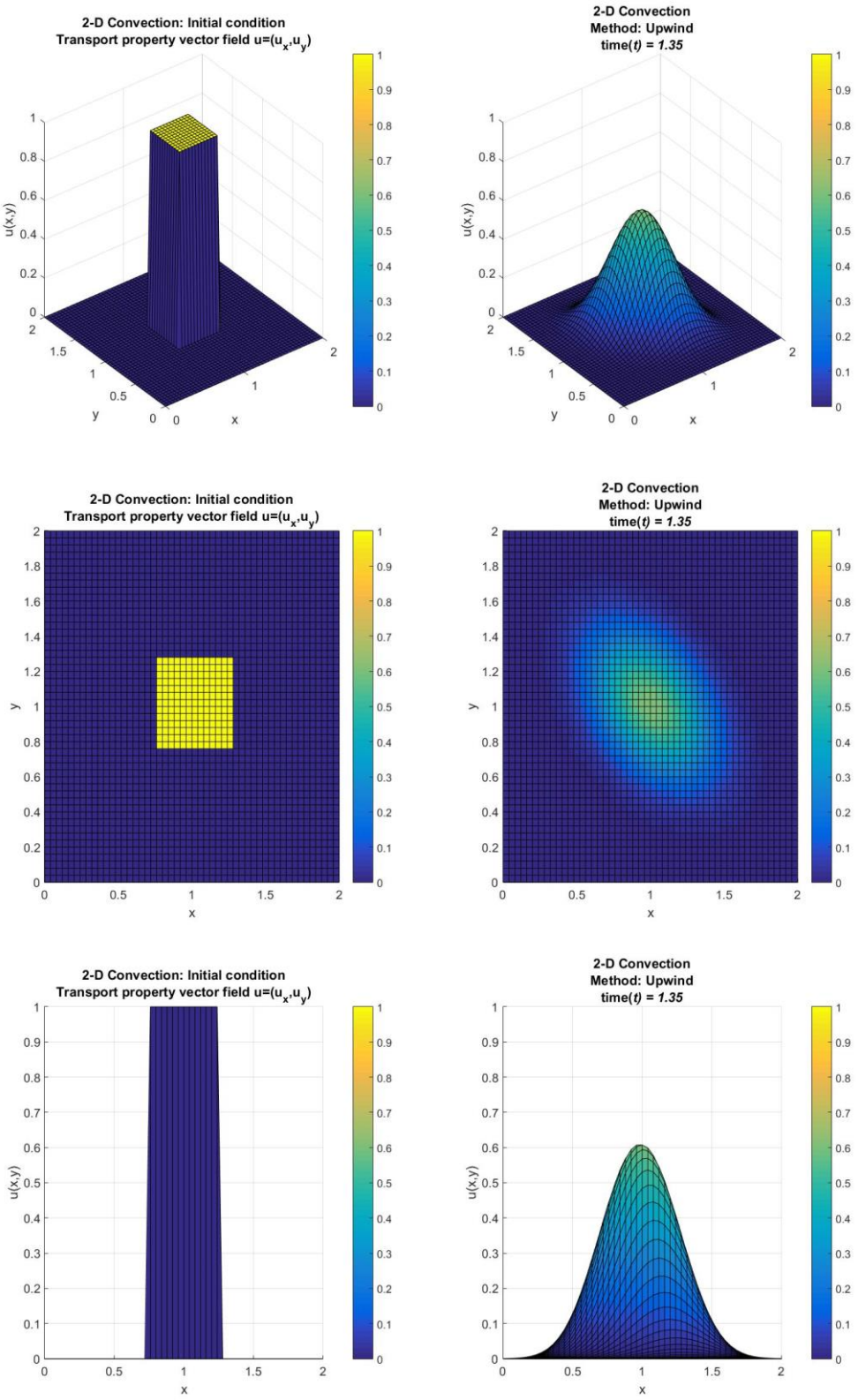


Figure 28. 2D Square wave advected by Upwind method

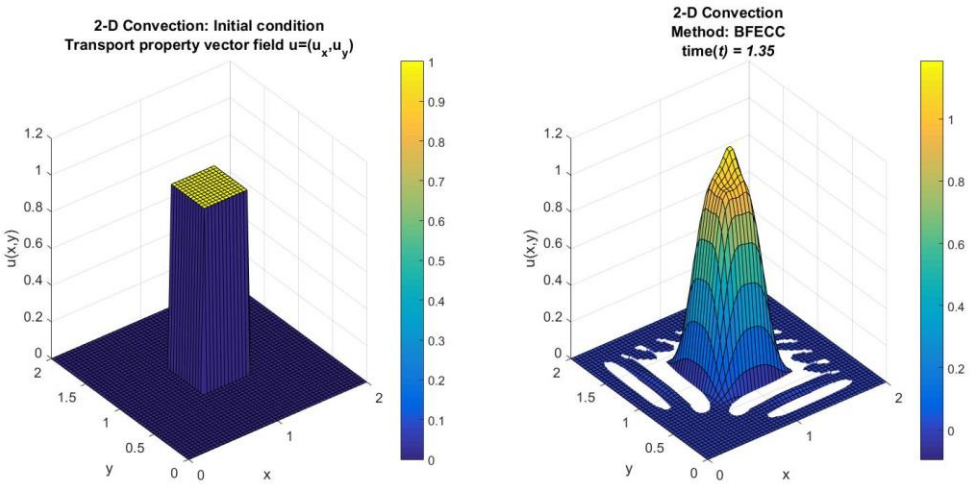
Finally, the non-smooth square wave function is solved using the BFECC method. The BFECC solutions shows, as expected, numerical oscillations on both the top and the bottom edges of the function. After one loop in the domain, the convection of the square wave with a velocity of 1.5 in each direction, a time step of 0.01 and a grid spacing of 0.04, the overshooting is about 19.4% on the top and the undershooting is about 9.9% on the bottom. These numerical oscillations cause solution values out of the domain of the function, which needs to be corrected improving the methodology.

CFL x direction	**CFL y direction**	**Method**	**Overshooting**	**Undershooting**
0.75	0.75	BFECC	19.373 %	9.872%

Table 13. 2D Square wave advected by BFECC method

From the figures one can observe that the numerical oscillations lead to values greater than 1 on the top edge and smaller than 0 on the bottom edge.

Even though, the solution of the BFECC seems to be much better than the Upwind solution, since at least the BFECC can better preserve the initial shape. It has to be considered that this numerical test is an extreme example far from the reality and wants to take the BFECC to the limit to identify the drawbacks of the method.



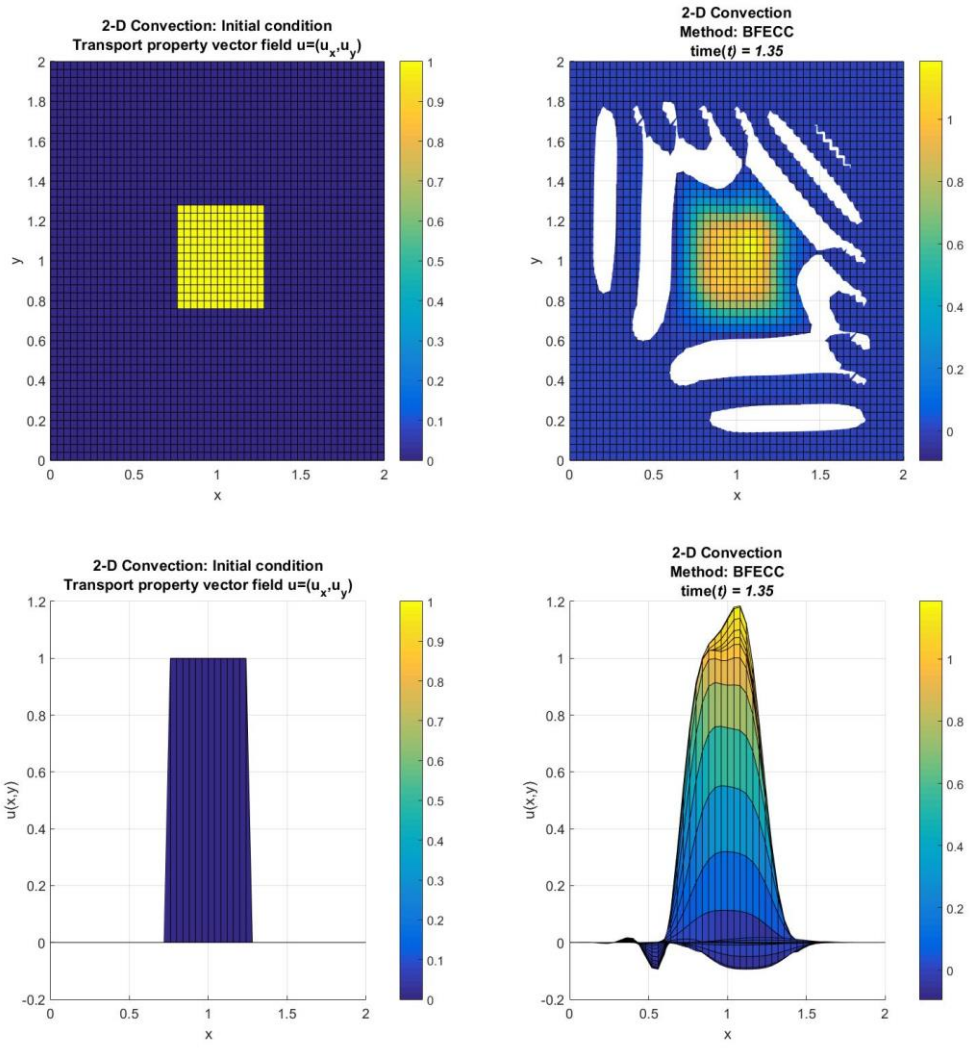


Figure 29. 2D Square wave advected by BFECC method

In summary, in the 2D numerical test examples, one can conclude that the Upwind method does not show numerical oscillations despite the smooth level of the function. On the other hand, its solution appears to excessively diffuse. The BFECC method behaves properly when dealing with smooth functions, but its solution locally oscillates in the steep zones of non-smooth functions.

5.3 Kratos numerical test example in 2D

In this section a numerical test example is performed in Kratos in order to test the accuracy and stability of the solution with the current algorithm for solving convection equations, that is the BFECC method. The numerical test performed will consist on solving a pure convection problem, set up by a heat focus on a rotational velocity field. Basically the partial differential equation that is solved in this problem is the following:

$$\frac{\partial T(\boldsymbol{x},t)}{\partial t} + a_x(\boldsymbol{x})\frac{\partial T(\boldsymbol{x},t)}{\partial x} + a_y(\boldsymbol{x})\frac{\partial T(\boldsymbol{x},t)}{\partial y} = 0 \tag{69}$$

where T is the temperature that will be transported over time t along the $\boldsymbol{x}$ spatial coordinates. The heat focus will be moved by the non-constant velocity field $\boldsymbol{a}(\boldsymbol{x}) = (a_x, a_y)$.

The mesh is generated by the pre and post processer GiD and is formed by linear triangular elements in an unstructured disposition.

The heat focus is the initial condition of the transient partial differential equation and has the following shape:

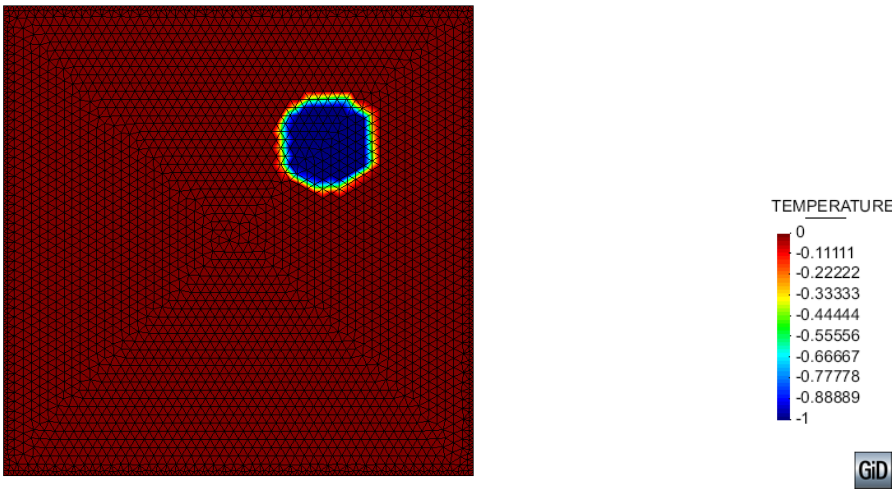


Figure 30. Initial condition for Kratos test example

As commented above, the heat focus will be moved by a rotating velocity field along the square domain. The absolute value of the velocity vector that will move the heat focus is 0.45 in this example:

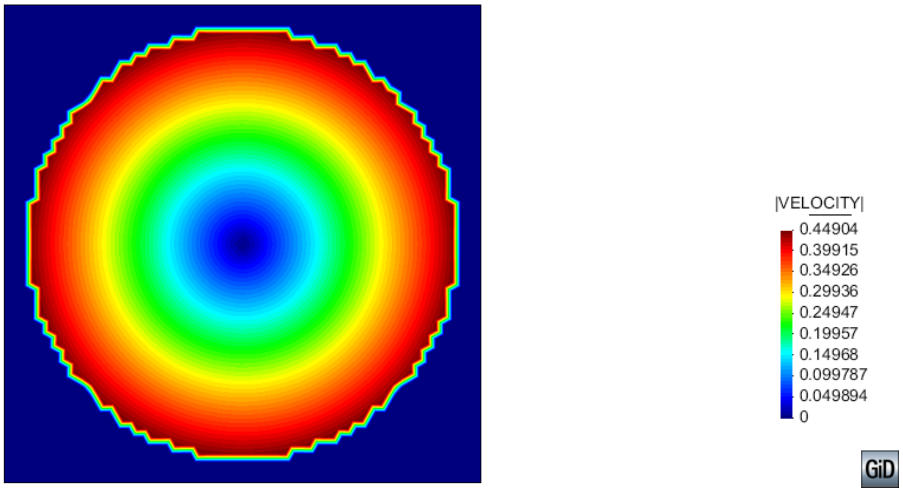


Figure 31. Velocity field for Kratos test example

Specifically, the code of the velocity field defined in the Kratos test example is the following:

```
#assign velocity field
import math
xc = 0.7 - 0.5
yc = 0.7 - 0.5
for node in base_model_part.Nodes:
    xlocal = node.X - 0.5
    ylocal = node.Y - 0.5
    r = math.sqrt( (xlocal)**2 + (ylocal)**2 )
    if( r < 0.45):
        node.SetSolutionStepValue(VELOCITY_X,0,-ylocal)
        node.SetSolutionStepValue(VELOCITY_Y,0,xlocal)

        d = math.sqrt( (xlocal - xc)**2 + (ylocal-yc)**2 ) - 0.1

        node.SetSolutionStepValue(TEMPERATURE,0,d)
        if(d <= 0.0):
            node.SetSolutionStepValue(TEMPERATURE,0,-1.0)
        else:
            node.SetSolutionStepValue(TEMPERATURE,0,0.0)
```

Figure 32. Velocity field's code for Kratos test example

The test is performed with a final time of 10 sec and a time stepping of 0.05 sec. That corresponds to 200 time steps. So, in the numerical test, the heat focus rotates a bit more than one loop. The solution of the BFECC method implemented in Kratos after one loop, that is at time step 127 corresponding to 6.35 sec, has the following values:

	Time step	**Initial condition**	**BFECC**	**Overshooting/ undershooting**
Minimal temperature	127	-1	-1.2048	20.48%
Maximal temperature	127	0	0.12279	12.28%

Table 14. Numerical solution after one loop

The numerical oscillations appear in the Kratos solution using the BFECC method. Specifically, the focus is amplified up to 20% and outside the focus limit the solutions seems to oscillate around 12%.

The following figure shows the Kratos solution using the BFECC method after one loop of the heat focus.

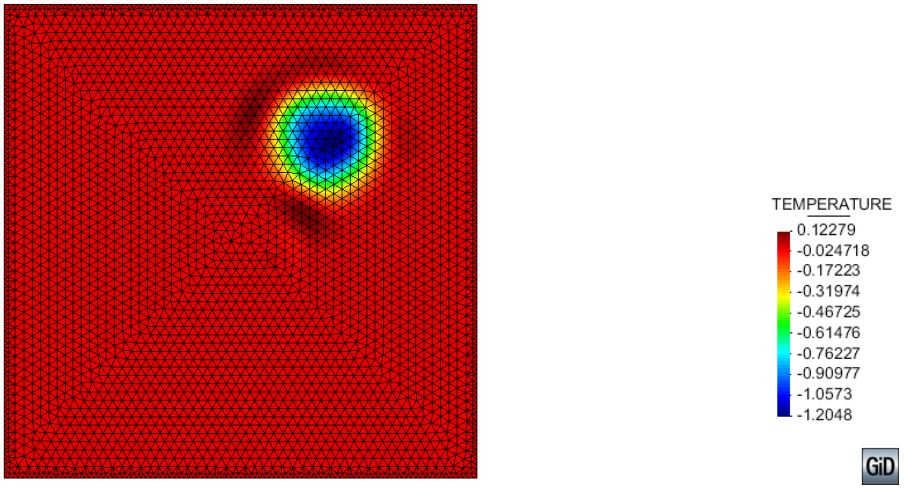


Figure 33. Kratos' solution with BFECC method after one loop

From this point of view is difficult to see where exactly the numerical oscillations appear, so limiting the maximum value to 0, the following figure identifies (in black) the zones where the solution is greater than 0. It can be seen that the numerical oscillations appear in a vast area of the domain. The numerical oscillations not only surround the heat focus but also appear far from it.

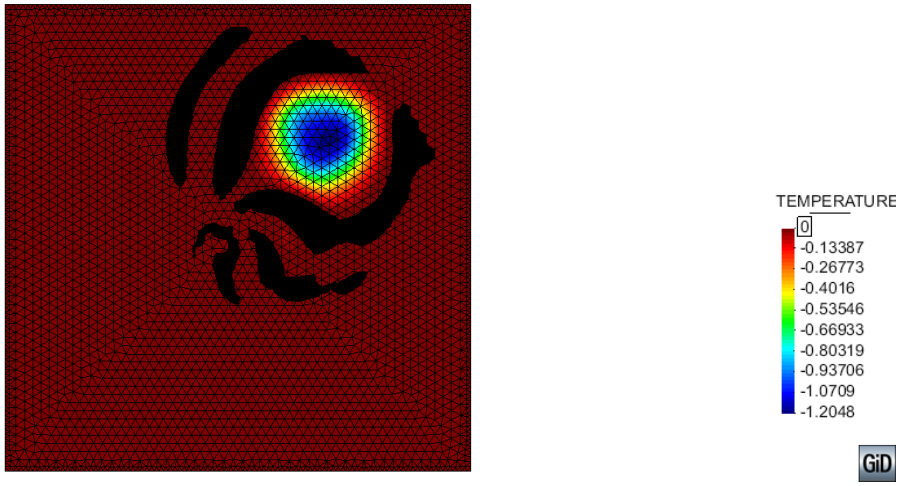


Figure 34. Numerical oscillations with BFECC method after one loop

Far from disappear or even maintaining, the numerical oscillations keep growing with time. At the final time step of the numerical test (after 10 seconds of moving the heat focus), that is at time step equal to 200, the solution computed with Kratos using the BFECC method is the following:

	Time step	**Initial condition**	**BFECC**	**Overshooting/ undershooting**
Minimal temperature	200	-1	-1.2323	23.23%
Maximal temperature	200	0	0.13222	13.22%

Table 15. Numerical oscillations at final time step

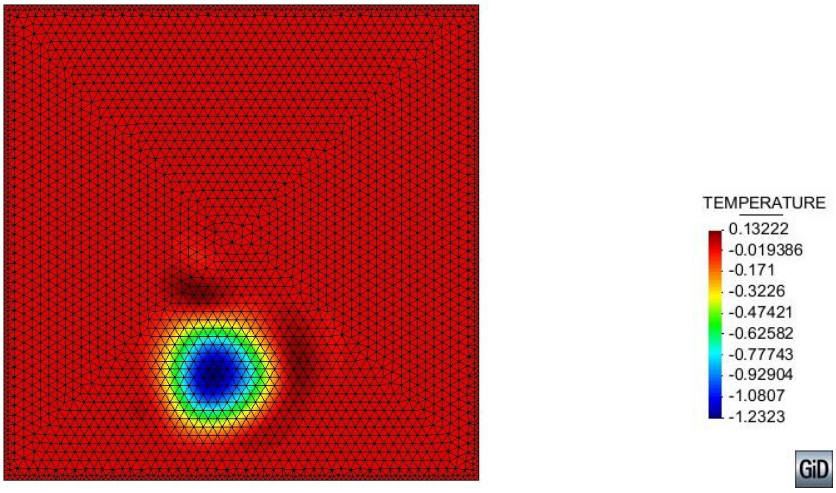


Figure 35. Kratos' solution with BFECC method at final time step

At the final time of the numerical test, the solution shows large numerical oscillations. In the centre of the heat focus the solution has an overshooting of 23.23%, that is that the temperature has decreased from -1ºC to -1.23ºC, and outside the heat focus the solution has an undershooting of 13.22%, so the temperature has increased from 0ºC to 0.13ºC.

For the sake of clarity, the temperature has been limited as in the first loop solution. In the following picture can be easily identified the location of the numerical oscillations that the BFECC solution perform.

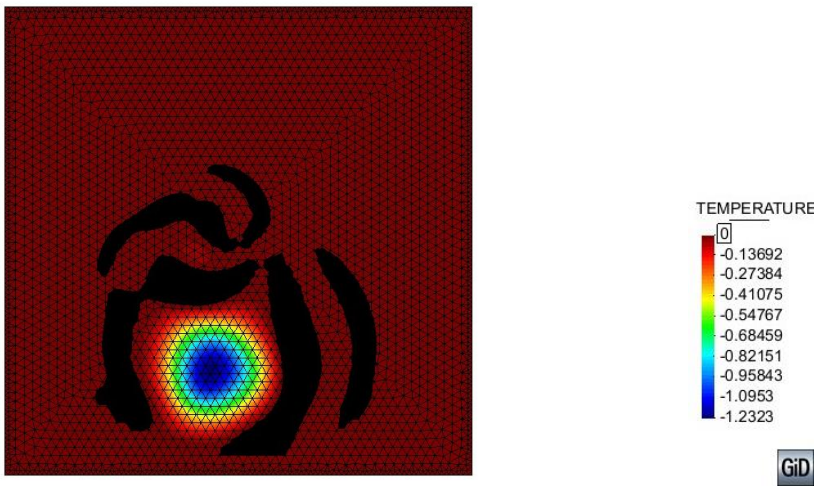


Figure 36. Numerical oscillations with BFECC method at final time step

All in all, Kratos can solve a pure convection problem using the BFECC method but performing large numerical oscillations when dealing with strongly non-smooth functions. It has been observed that the oscillations appear both in the focus that is being moved and also outside the focus, where the solution should not be affected by the movement of the focus.

The perfect solution should be the one performed by a methodology capable of preserving the original shape all over the simulation, eliminating the spurious numerical oscillations that can fake the final results.

The succeeding chapter deals with the implementation of the limiting strategy able to eliminate the numerical oscillations in the current BFECC method implemented in Kratos.

CHAPTER 6

Computer implementation

The following chapter contains the detailed explanation of the implementation of the shock-capturing strategy in Kratos solver. First of all, the actual variables of the code currently implemented in Kratos that solves advection equations applying the BFECC method are described. Then, the main routine of the implementation of the shock-capturing strategy in the BFECC method is explained step by step. Finally, it is presented another way to implement the shock-capturing strategy, not in the elements nodes but in the barycentre of the elements. This way of proceeding has several advantages that will be commented in the corresponding section.

6.1 Algorithm implementation in Kratos

The implementation of the shock-capturing strategy in Kratos has been made modifying the original Kratos code that solves convection problems. For the sake of clarity, Table 16 contains de relation between the variable's names of the BFECC with shock-capturing strategy algorithm developed in 4.6 and the variable's names in Kratos code.

The `FastGetSolutionStepValue` and `GetValue` arrays are already used in Kratos and simply define auxiliary arrays where the position of the particles are stored. For time step t^n these auxiliary arrays save the position in `(rVar,1)` and in time step t^{n+1} the position is stored in `(rVar)`.

Algorithm steps and variables	**Description**	**Kratos code**
φ^n	Original particle's position	`FastGetSolutionStepValue(rVar,1)`
$\hat{\varphi}^{n+1}$	Particle's position at time step $n+1$	Computed as `phi1` and stored in `FastGetSolutionStepValue(rVar)`
$\hat{\varphi}^n$	Particle's position at time step n after moving it forward and backward	`phi_old`
$e^{(1)}$	BFECC error	`GetValue(ERROR_1)`
$\tilde{\varphi}^n$	Corrected original particle's position	`GetValue(rVar)`
θ^{n+1}	Solution of BFECC method	Computed as `phi1` and stored in `FastGetSolutionStepValue(rVar)`
$\mathcal{B}(\theta^{n+1})$	Backward advection of BFECC solution	`phi2`
$e^{(2)}$	Comparative error of shock-capturing strategy	`GetValue(ERROR_2)`
$\tilde{e}^{(1)}$	Modification of the BFECC error	`GetValue(ERROR)`
j grid points	Adjacent grid points to every i grid point in the mesh	`adj_iparticle`
$\varphi^n + \tilde{e}^{(1)}$	Modification of the original particle's position in such nodes where numerical oscillations occur	`GetValue(rVar)`
φ^{n+1}	Particle's position at time step $n+1$ applying the shock-capturing strategy in the BFECC method	Computed as `phi1` and stored in `FastGetSolutionStepValue(rVar)`

Table 16. Algorithm variable's name in Kratos code

6.2 Interpolation strategy

The BFECC algorithm is based on first convecting the particles in forward or backward sense, look for the position of the moved particle and then do a interpolation in order to find the value of the solution from the known values of the neighbour nodes. This interpolation is done following the so called *trilinear interpolation* for three dimensional problems or *bilinear interpolation* for two dimensional problems, see [12] for more information about interpolation techniques.

Interpolation is defined as the operation of constructing new data from the information that is currently available. In the BFECC method, the interpolation will consist on computing the particle's position from the different nodes that form the element. Then we will be able to know the contribution of each element node to the intermediate point.

Specifically in 2D, bilinear interpolation approximates the value of an intermediate point within the local axial element linearly, using data on the lattice points. It is an extension of linear interpolation for interpolating functions of two variables on a 2D grid. The key idea is to perform linear interpolation first in one direction and then again in the other direction. Although each step is linear in the sampled values and in the position, the interpolation as a whole is not linear but rather quadratic in the sample location. An example of a bilinear interpolation can be found in Figure 37. It consists on a unit regular square with known values at the lattice points of the element and the interpolation all over the domain in a colour scale.

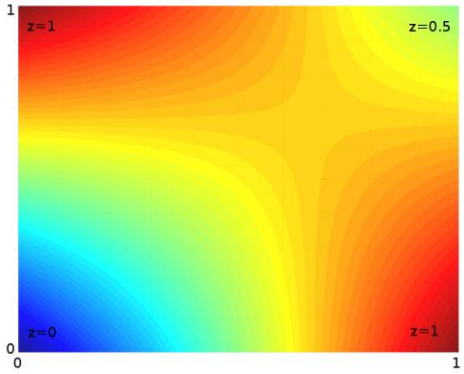


Figure 37. Bilinear interpolation

The extension of bilinear interpolation in three dimensions is the trilinear interpolation as commented before. These algorithm is especially useful since it allows us to compute the value of a point inside a cube from the values of the vertices of such cube as is shown in Figure 38.

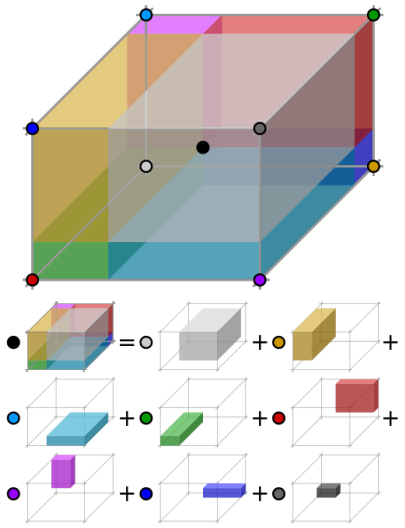


Figure 38. Trilinear interpolation

6.3 BFECC with shock-capturing strategy routine in Kratos

The code currently implemented in Kratos that solves convection problems is based on the BFECC method. In this essay, the original code has been modified in order to take into account the limiting strategy explained in 4.6. The routine described below is the set of functions that contain the algorithm of BFECC with the shock-capturing strategy method. The routine commented below is located in the *bfecc_limiter_convection.h* file from the Kratos' *convection diffusion application* and it is programmed in *C++*.

6.3.1 Main routine

First of all, the implemented code gets the information from a *ModelPart* previously created with the pre and post processor GiD. The *ModelPart* (.mdpa) is a file that Kratos can understand and it contains information about the definition of the geometry of the element, the material properties, the loads applied, the boundary conditions imposed, etc.

The mesh of the model part defines elements, nodes and conditions. The mesh size (number of nodes) is stored in the *nparticles* variable. The code will go through each of these points to compute the problem solution. Then the code will store the physical substance transported as the *rVar* variable. In this chunk, the information related with the time step used to solve the problem is read and stored in the *dt* variable. There is also the necessity of creating the arrays where the information regarding the elements will be placed, as well as an estimation of the maximum number of results that we may obtain of the problem solved.

```
void BFECCconvect(ModelPart& rModelPart, const Variable< double >& rVar, const
Variable<array_1d<double,3> >& conv_var, const double substeps)
{
KRATOS_TRY
const double dt = rModelPart.GetProcessInfo()[DELTA_TIME];

array_1d<double, TDim + 1 > N;
const int max_results = 10000;
typename BinBasedFastPointLocator<TDim>::ResultContainerType results(max_results);

const int nparticles = rModelPart.Nodes().size();

PointerVector< Element > elem_backward( rModelPart.Nodes().size());
std::vector< array_1d<double,TDim+1> > Ns( rModelPart.Nodes().size());
std::vector< bool > found( rModelPart.Nodes().size());
std::vector< bool > foundf( rModelPart.Nodes().size());
```

Figure 39. Processing information from the *ModelPart*

Once all the information from the problem is stored, the code starts implementing the BFECC method. The first step is the backward interpolation. To do so, the code goes through all nodes using a loop from 0 to the number of nodes (*nparticles*). The backward of the BFECC needs to interpolate between the solution at the current node and the solution at the node found after applying the *ConvectBySubstepping* function. This function is used to find the position of the node after the backward step. This function does not move the particles but does move all the field in the contrary sense of the velocity of the particles. That is, when we want to move the particles in a forward sense, the code moves all the temperature field in a backward sense. The element containing the backward advected particle is stored in `elem_backward(i)`. We will use this location later on. If the position of the particle convected is found, meaning that it is located inside the domain, the code interpolates the solution as:

```
phi1 = N[0]*(geom[0].FastGetSolutionStepValue(rVar,1));
```

```
phi1 += N[k]*(geom[k].FastGetSolutionStepValue(rVar,1));
```

where N are the shape functions used in the interpolation technique introduced in 6.2, the *(rVar,1)* is the solution of property transported in the previous step n and *phi1* is the solution at time $n+1$. After that, this solution is stored as:

```
iparticle->FastGetSolutionStepValue(rVar) = phi1;
```

```
//STEP 1: Backward advection to estimate rVar(N+1) from N
#pragma omp parallel for firstprivate(results,N)
for (int i = 0; i < nparticles; i++) {
      typename BinBasedFastPointLocator<TDim>::ResultIteratorType result_begin = ↙
      results.begin();

      ModelPart::NodesContainerType::iterator iparticle=rModelPart.NodesBegin()+i;
```

```
        Element::Pointer pelement;
        array_1d<double,3> bckPos = iparticle->Coordinates();
        const array_1d<double,3>& vel=iparticle->FastGetSolutionStepValue(conv_var);
        bool is_found = ConvectBySubstepping(dt,bckPos,vel, N, pelement, ↙
        result_begin, max_results, -1.0, substeps);
        found[i] = is_found;

        if(is_found) {
                //save position backwards
                elem_backward(i) = pelement;
                Ns[i] = N;

                Geometry< Node < 3 > >& geom = pelement->GetGeometry();
                double phi1 = N[0] * ( geom[0].FastGetSolutionStepValue(rVar,1));
                for (unsigned int k = 1; k < geom.size(); k++) {
                        phi1 += N[k] * ( geom[k].FastGetSolutionStepValue(rVar,1) );
                }
                iparticle->FastGetSolutionStepValue(rVar) = phi1;
        }
}
```

Figure 40. Backward advection

The second step in the BFECC method is to obtain the solution at the previous time step n once we moved the points in a forward sense. To do so, a forward interpolation is made with the position of the particles after moving them applying the *ConvectBySubstepping* function with positive velocity. For each node in the mesh, the code computes the solution of the substance transported at time n as:

```
phi_old = N[0] * (geom[0].FastGetSolutionStepValue(rVar));
phi_old  += N[k] * (geom[k].FastGetSolutionStepValue(rVar));
```

The solution is stored as *phi_old*, and will be used to compute the BFECC error comparing it with the initial value at time n.

```
//STEP 2: Forward advection to obtain rVar(N) from N+1
#pragma omp parallel for firstprivate(results,N)
for (int i = 0; i < nparticles; i++) {
        typename BinBasedFastPointLocator<TDim>::ResultIteratorType result_begin = ↙
        results.begin();

        ModelPart::NodesContainerType::iterator iparticle=rModelPart.NodesBegin()+i;

        Element::Pointer pelement;
        array_1d<double,3> fwdPos = iparticle->Coordinates();
        const array_1d<double,3>& vel = iparticle-> ↙
        FastGetSolutionStepValue(conv_var,1);
        bool is_found = ConvectBySubstepping(dt,fwdPos,vel, N, pelement, ↙
        result_begin, max_results, 1.0, substeps);
        foundf[i] = is_found;

        if(is_found) {
                Geometry< Node < 3 > >& geom = pelement->GetGeometry();
                double phi_old = N[0] * ( geom[0].FastGetSolutionStepValue(rVar));
```

```
            for (unsigned int k = 1; k < geom.size(); k++) {
                  phi_old  += N[k] * ( geom[k].FastGetSolutionStepValue(rVar) );
            }
```

Figure 41. Forward advection

We are now able to compute the BFECC error. We are going to call this error $e^{(1)}$, because it will be compared with another error lately. This error is computed with the value of the initial solution at time n and the value of the solution at time n after the backward and forward steps as:

$$e^{(1)} = \frac{1}{2}(\varphi^n - \hat{\varphi}^n) \tag{70}$$

where φ^n is the original solution and $\hat{\varphi}^n$ is the solution after the backward and forward advection steps, that is *phi_old* in the code.

Then, this error is summed up to the initial solution at time n, modifying the initial value of the substance transported. This value is stored in the *GetValue* variable at time step n.

```
//Computing error 1 and modified solution at time N to be interpolated again
iparticle->GetValue(ERROR_1) = 0.5*iparticle->FastGetSolutionStepValue(rVar,1) - ↙
0.5*phi_old;

iparticle->GetValue(rVar) = iparticle->FastGetSolutionStepValue(rVar,1) + ↙
iparticle->GetValue(ERROR_1);//rVar(n)+e1
      }
}
```

Figure 42. Computing the error of the BFECC method

The final step of the BFECC method is to interpolate again the modified solution in time step n in order to get the solution at time step $n+1$ using a backward interpolation. The interpolation is made overwriting the previous solution:

```
phi1 += N[k] * ( geom[k].GetValue(rVar) );
```

And will be stored in the `FastGetSolutionStepValue` array. In this step we use the location of the points found in the first backward step. That is why the *ConvectBySubstepping* function is not necessary here. We previously saved the element after the backward advection was made in `elem_backward(i)` for each node.

```
//STEP 3: Backward advection with modified solution
#pragma omp parallel for
for (int i = 0; i < nparticles; i++) {
      ModelPart::NodesContainerType::iterator iparticle= rModelPart.NodesBegin()+i;
      bool is_found = found[i];
      if(is_found) {
            array_1d<double,TDim+1> N = Ns[i];
```

```
                Geometry< Node < 3 > >& geom = elem_backward[i].GetGeometry();
                double phi1 = N[0] * ( geom[0].GetValue(rVar));
                for (unsigned int k = 1; k < geom.size(); k++) {
                        phi1 += N[k] * ( geom[k].GetValue(rVar) );
                }
                iparticle->FastGetSolutionStepValue(rVar) = phi1;
        }
}
```

Figure 43. Backward advection with modified solution

By this time, the solution obtained is the one of the BFECC method. The modification and improvement of the code starts here, introducing another error $e^{(2)}$ that will identify the oscillations produced by the simple BFECC method when it is compared with $e^{(1)}$.

As shown in 4.6, the $e^{(2)}$ is computed as:

$$e^{(2)} = \varphi^n - \left(\mathcal{B}(\theta^{n+1}) + e^{(1)}\right) \tag{71}$$

where θ^{n+1} is the solution obtained in the previous step, that is the solution of the BFECC method.

In this step, the solution is first advected again to obtain the solution at time step n. The advection in this step is done in a forward sense, seeking the position of the particles with the *ConvectBySubstepping* function with positive velocity. The interpolation in forward sense is stored as:

```
phi2 += N[k] * ( geom[k].FastGetSolutionStepValue(rVar) );
```

After that, the comparative error $e^{(2)}$ is computed using the initial solution (`FastGetSolutionStepValue(rVar,1)`), with the forward advection of the previous solution (`phi2`) and the BFECC error (`GetValue(ERROR_1)`).

```
//STEP 4: Computing error 2 with forward of solution at N+1
#pragma omp parallel for firstprivate(results,N)
for (int i = 0; i < nparticles; i++) {
        typename BinBasedFastPointLocator<TDim>::ResultIteratorType result_begin = ↙
        results.begin();

        ModelPart::NodesContainerType::iterator iparticle=rModelPart.NodesBegin()+i;

        Element::Pointer pelement;
        array_1d<double,3> fwdPos = iparticle->Coordinates();
        const array_1d<double,3>& vel= iparticle->FastGetSolutionStepValue(conv_var);
        bool is_found = ConvectBySubstepping(dt,fwdPos,vel, N, pelement, ↙
        result_begin, max_results, 1.0, substeps);
        found[i] = is_found;

        if(is_found) {
                //Forward
                Geometry< Node < 3 > >& geom = pelement->GetGeometry();
                double phi2 = N[0] * ( geom[0].FastGetSolutionStepValue(rVar));
                for (unsigned int k = 1; k < geom.size(); k++) {
```

```
                    phi2 += N[k] * ( geom[k].FastGetSolutionStepValue(rVar) );
                }
                //Computing e2
                iparticle->GetValue(ERROR_2) = iparticle -> ↙
                FastGetSolutionStepValue(rVar,1)-(phi2+iparticle->GetValue(ERROR_1));
        }
}
```

Figure 44. Computing the comparative error

The shock-capturing strategy consists on comparing the two errors to look for numerical oscillations. As explained in 4.6, the $e^{(1)}$ will be modified in the neighbour nodes of the node that display numerical oscillations. So, the first step is to identify the neighbour nodes in the mesh for every node. This is made by the process `FindNodalNeighboursProcess` already implemented in Kratos. After that, we define a copy of the $e^{(1)}$, named $\tilde{e}^{(1)}$, and this will be the final error used to correct the numerical oscillations. In the code, the $e^{(1)}$ corresponds to `ERROR_1` and the $\tilde{e}^{(1)}$ corresponds to `ERROR`.

```
iparticle->GetValue(ERROR) = iparticle->GetValue(ERROR_1);
```

```
// STEP 5: Shock-capturing strategy (1)
FindNodalNeighboursProcess adj_process(rModelPart, 10, 10);
adj_process.ClearNeighbours();
adj_process.Execute();

#pragma omp parallel for
for (int i = 0; i < nparticles; i++) {
        ModelPart::NodesContainerType::iterator iparticle=rModelPart.NodesBegin()+i;
        iparticle->GetValue(ERROR) = iparticle->GetValue(ERROR_1);
}
```

Figure 45. Finding the nodes of the neighbour elements

Essentially, the shock-capturing strategy consists on modifying the error of the neighbour nodes (j) used to interpolate the solution in the node where the numerical oscillation occur, that is when the $|e_i^{(2)}| \leq |e_i^{(1)}|$ condition is violated. So, after identifying the neighbour nodes of every particle, the code locates the nodes where numerical oscillations occur with the `if()` condition. Then, for the i nodes where this condition is not fulfilled, the code modifies the error of the j neighbour particles with the *minmod* function. Basically, the *minmod* function assigns the minimum absolute value between $e_i^{(1)}$ and $\tilde{e}_j^{(1)}$ to the j node, not to the i node.

```
// STEP 5: Shock-capturing strategy (2)
#pragma omp parallel for
for (int i = 0; i < nparticles; i++) {
        ModelPart::NodesContainerType::iterator iparticle=rModelPart.NodesBegin()+i;
        WeakPointerVector<Node<3> > & adj_iparticle = iparticle-> ↙
        GetValue(NEIGHBOUR_NODES);

        if(std::abs(iparticle->GetValue(ERROR_2)) > std::abs(iparticle-> ↙
        GetValue(ERROR_1))){
```

```
            for(int j = 0 ; j < iparticle->GetValue(NEIGHBOUR_NODES).size(); j++) {
                  adj_iparticle[j].GetValue(ERROR) = minmod(iparticle-> ↙
                  GetValue(ERROR_1),adj_iparticle[j].GetValue(ERROR));
            }
      }
}
```

Figure 46. *minmod* limiter where numerical oscillations occur

This correction of the error in the neighbour nodes of the particles where numerical oscillations occur, is then used to modify the original particle's position as $\varphi^n + \tilde{e}^{(1)}$ at time step n.

```
// STEP 5: Shock-capturing strategy (3)
#pragma omp parallel for
for (int i = 0; i < nparticles; i++) {
      ModelPart::NodesContainerType::iterator iparticle=rModelPart.NodesBegin()+i;
      bool is_found = foundf[i];
      if(is_found) {
            iparticle->GetValue(rVar)=iparticle->FastGetSolutionStepValue(rVar,1)↙
            + iparticle->GetValue(ERROR);
      }
}
```

Figure 47. New correction of the solution at time step n

At this point, the shock-capturing strategy is implemented and has been used to correct the original particle's position improving the BFECC strategy, which modified the original particle's position without taking into account the numerical oscillations that may appear in non-smooth functions. So the final step is shared by both methods and it consists on interpolating the corrected solution at time step n to get the right solution at time step $n+1$. The interpolation is done applying the same shape functions that were used to compute the comparative errors moving the particles forward and backward. Again, the application of the *ConvectBySubstepping* function is not necessary here due to the fact that the position of the particles where already stored in the first steps of the process and has not changed. The interpolation leads to the final solution of the method `phi1` that corresponds to φ^{n+1}:

```
//STEP 6: Backward with corrected solution at points with numerical oscillations
#pragma omp parallel for
for (int i = 0; i < nparticles; i++){

      ModelPart::NodesContainerType::iterator iparticle=rModelPart.NodesBegin()+i;

      bool is_found = found[i];
      if(is_found) {
            array_1d<double,TDim+1> N = Ns[i];
            Geometry< Node < 3 > >& geom = elem_backward[i].GetGeometry();
            double phi1 = N[0] * ( geom[0].GetValue(rVar));
            for (unsigned int k = 1; k < geom.size(); k++) {
                  phi1 += N[k] * ( geom[k].GetValue(rVar) );
            }
```

```
                iparticle->FastGetSolutionStepValue(rVar) = phi1;
        }
}
```

Figure 48. Backward advection with corrected solution

6.3.2 Complementary functions

As seen in the previous section, the main routine requires some functions that are programmed out of the main routine. We are talking about the *minmod* function, used in the limiting strategy, and the *ConvectBySubstepping*, used in the advection process.

The *minmod* function has been described previously in 4.6 and essentially looks for the minimum absolute value out of two scalars. It has been implemented for the shock-capturing strategy in such a way that the two scalars that we are comparing are represented by the `x` and `y` variables. Observe that if the two scalars have different sign the output of the function is a 0.

```
double minmod(double x, double y) {
double f;
        if(x > 0.0f && y > 0.0f)
                f = std::min(x,y);
        else if(x < 0.0f && y < 0.0f)
                f = std::max(x,y);
        else
                f = 0;
return f;
}
```

Figure 49. *minmod* function

The *ConvectBySubstepping* function already implemented in Kratos searches the position of the node after a forward or a backward step. The sign of the velocity (*velocity_sign*) introduced in the function indicates whether it looks for previous position or post position. As commented before, the function does not move the particles but does move all the field in the contrary sense of the velocity of the particles. So if the particles need to be moved in a forward sense, the code moves all the field in a backward sense.

The *substepping* strategy simply consists on convecting the particles not in one step corresponding to the time interval defined (*dt*) but with several smaller steps (*small_dt*) for increasing the accuracy of the convection procedure.

```
bool ConvectBySubstepping(
        const double dt,
        array_1d<double,3>& position, //IT WILL BE MODIFIED
        const array_1d<double,3>& initial_velocity,
        array_1d<double,TDim+1>& N,
        Element::Pointer& pelement,
        typename BinBasedFastPointLocator<TDim>::ResultIteratorType& result_begin,
```

```
       const unsigned int max_results,
       const double velocity_sign,
       const double subdivisions) {
bool is_found = false;
array_1d<double,3> veulerian;
const double small_dt = dt/subdivisions;

if(velocity_sign > 0.0) //going from the past to the future {
       noalias(position) += small_dt*initial_velocity;
       unsigned int substep = 0;
       while(substep++ < subdivisions) {
              is_found = mpSearchStructure->FindPointOnMesh(position, N, pelement, ↙
              result_begin, max_results);

              if (is_found == true) {
                     Geometry< Node < 3 > >& geom = pelement->GetGeometry();

                     const double new_step_factor = ↙
                     static_cast<double>(substep)/subdivisions;
                     const double old_step_factor = (1.0 - new_step_factor);

                     noalias(veulerian) = N[0] * ↙
                     (new_step_factor*geom[0].FastGetSolutionStepValue(VELOCITY) + ↙
                     old_step_factor*geom[0].FastGetSolutionStepValue(VELOCITY,1));
                     for (unsigned int k = 1; k < geom.size(); k++) {
                        noalias(veulerian)+=N[k]*(new_step_factor* ↙
                        geom[k].FastGetSolutionStepValue(VELOCITY)+old_step_factor* ↙
                        geom[k].FastGetSolutionStepValue(VELOCITY,1) );

                        noalias(position) += small_dt*veulerian;

                     }
              else
                     break;
              }
       }
else //going from the future to the past
{
       noalias(position) -= small_dt*initial_velocity;
       unsigned int substep=0;
       while(substep++ < subdivisions) {
              is_found = mpSearchStructure->FindPointOnMesh(position, N, pelement, ↙
              result_begin, max_results);

              if (is_found == true) {
                     Geometry< Node < 3 > >& geom = pelement->GetGeometry();

                     //this factors get inverted from the other case
                     const double old_step_factor = ↙
                     static_cast<double>(substep)/subdivisions;
                     const double new_step_factor = (1.0 - old_step_factor);

                     noalias(veulerian) = N[0] * ↙
                     (new_step_factor*geom[0].FastGetSolutionStepValue(VELOCITY) + ↙
                     old_step_factor*geom[0].FastGetSolutionStepValue(VELOCITY,1));
```

```
                    for (unsigned int k = 1; k < geom.size(); k++)
                       noalias(veulerian) += N[k] *(new_step_factor* ↙
                       geom[k].FastGetSolutionStepValue(VELOCITY)+old_step_factor* ↙
                       geom[k].FastGetSolutionStepValue(VELOCITY,1));

                       noalias(position) -= small_dt*veulerian;

                    }
              else
                    break;
              }
       }

       return is_found;

}
```

Figure 50. *ConvectBySubstepping* function

6.4 One step further: application of the shock-capturing strategy at the barycentre of the elements

The BFECC with shock-capturing strategy can be also applied in Finite Elements (FEM) codes. Due to the structure of the data bases of the FEM codes, it may be interesting to evaluate the errors not in the nodes of the element but in the barycentre of the elements. To do so, the following code has been implemented.

The first steps corresponding to the BFECC method are exactly the same as in the previous commented code. The difference is located in the shock-capturing strategy. Essentially, the strategy maintains the same layout, but the nodes where the errors are corrected are different.

In the previous case, where the shock-capturing was applied in the elements nodes, the limiting strategy, so the comparison between error, took place between adjacent grid nodes. The strategy consist on modifying the error in the adjacent nodes j of the node i where the numerical oscillations occur. The aim of this procedure is modifying the error of such nodes j that are used in the interpolation step of the solution in the node i, see Figure 51.

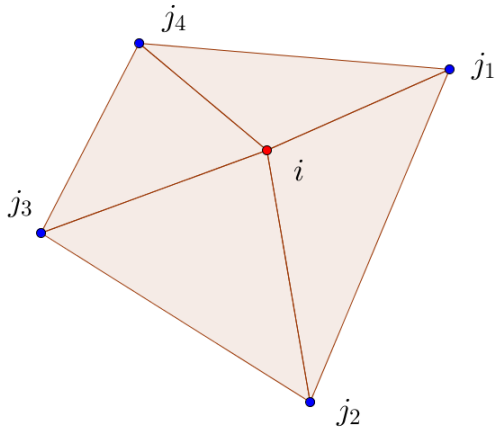


Figure 51. Limiting strategy at elements nodes

The other way of proceeding is to apply the limiting strategy, that is modifying the error, of nodes j not comparing them with the adjacent grid points, but doing it with the point placed in the barycentre of the element, see Figure 52. As commented, this procedure is very advantageous when working with finite element codes, because they use the barycentre point for many routines.

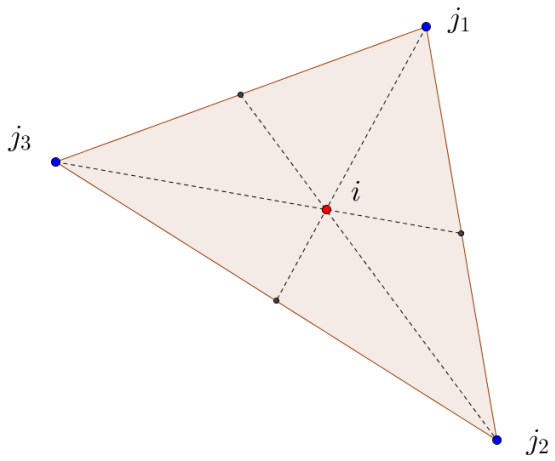


Figure 52. Limiting strategy at elements barycentre

First, the code is presented in detail and the methodology will be properly tested in posterior chapters.

The aim of this routine is to evaluate the solution in the centre of the element, as well as the errors $e^{(1)}$ and $e^{(2)}$. So, first of all, the code finds the position of the barycentre of each element as `GetGeometry().Center()` and evaluates the velocity also in the centre of the element.

```
// STEP 4 (1)
int nelements = rModelPart.NumberOfElements();
for(int i = 0 ; i < nelements; i++) {
```

```
        typename BinBasedFastPointLocator<TDim>::ResultIteratorType result_begin = ↙
        results.begin();

        ModelPart::ElementsContainerType::iterator i_element = ↙
        rModelPart.ElementsBegin() + i;
        Element::GeometryType& element_geometry = i_element->GetGeometry();

        Element::Pointer pelement;
        array_1d<double,3> fwdPos = i_element->GetGeometry().Center();
        array_1d<double,3> vel = ZeroVector(3);

        for(int j = 0 ; j < element_geometry.size(); j++) {
               for(int k = 0 ; k < 3; k++) {
                      vel[k]+=element_geometry[j].GetSolutionStepValue(conv_var)[k]/↙
                      element_geometry.size();
               }
        }
        bool is_found = ConvectBySubstepping(dt,fwdPos,vel, N, pelement, ↙
        result_begin, max_results, 1.0, substeps);//seeking forwards
        double e1 = 0.00f;

        for(int j = 0 ; j < element_geometry.size(); j++) {
               e1 += element_geometry[j].GetValue(ERROR_1);
        }
        e1 /= element_geometry.size();
```

Figure 53. Defining variables in the element's barycentre

Remind that for the computation of the comparative error $e^{(2)}$ is necessary the initial position of the particles. Since the objective now is to compute the comparative error in the barycentre of the elements, the initial solution that we are interested in is the position of the particle located in the barycentre (`solution_in_center`).

```
// STEP 4 (2)
        double e2 = e1;
        if(is_found) {
               //Forward with
               Geometry< Node < 3 > >& geom = pelement->GetGeometry();
               double phi2 = N[0] * ( geom[0].FastGetSolutionStepValue(rVar));
               for (unsigned int k = 1; k < geom.size(); k++) {
                      phi2 += N[k] * ( geom[k].FastGetSolutionStepValue(rVar) );
               }
               double solution_in_center = 0;
               //Computing error2 as e2 = rVar(n)-(phi2+e1)
               for(int j = 0 ; j < element_geometry.size(); j++) {
                      solution_in_center += i_element-> ↙
                      GetGeometry()[j].FastGetSolutionStepValue(rVar,1);
               }
               solution_in_center /= element_geometry.size();

               e2 = solution_in_center - (phi2 + e1);
```

Figure 54. Computing the comparative error in the barycentre

The next step that changes with respect to the routine in 6.3.1 is the shock-capturing strategy. The strategy preserves, but the nodes compared are different. Now we are interested in comparing errors that have been evaluated in the element's barycentre. Then, the neighbour nodes that need to be modified if numerical oscillations occur (that means the violation of the $|e_i^{(2)}| \leq |e_i^{(1)}|$ condition) are no longer nodes from neighbour elements, they are the nodes of each particular element that we are evaluating. So, the limiting strategy applies the *minmod* function in every element's node if the condition is violated in the centre of the element:

```
// STEP 5 (2)
              if(std::abs(e2) > std::abs(e1)) {
                     for(int j = 0 ; j < element_geometry.size(); j++){
                            element_geometry[j].GetValue(ERROR) = ↙
                            minmod(e1,element_geometry[j].GetValue(ERROR_1));
                     }
              }
       }
}
```

Figure 55. *minmod* limiter in the same element

After that, the same correction of the initial solution is done as in the 6.3.1 strategy. The difference is that the position that we are correcting here is the position of the particle located in the element's barycentre, from the comparison between the barycentre error and the nodal error of the same element.

```
// STEP 5 (3)
#pragma omp parallel for
for (int i = 0; i < nparticles; i++){
       ModelPart::NodesContainerType::iterator iparticle=rModelPart.NodesBegin()+i;
       bool is_found = foundf[i];
       if(is_found) {
              iparticle->GetValue(rVar)=iparticle->FastGetSolutionStepValue(rVar,1)↙
              + iparticle->GetValue(ERROR);
       }
}
```

Figure 56. Correcting nodal errors from each element

CHAPTER 7

Correction, validation and examples

The following chapter contains the assessment of the same test examples performed in Chapter 5 but now computing them with the new method, that is the BFECC method with the shock-capturing strategy acting as a limiter. The solution of the new method is going to be tested analysing the overshooting and undershooting of the one-dimensional examples as well as of the rotating heat focus test introduced in 5.3.

7.1 Improving the numerical solution for linear advection equation in 1D

The one-dimensional Matlab test examples used for testing the new methodology for solving advection equations are different kind of functions and CFL conditions. The functions and the CFL conditions are the same as the ones performed in 5.1, that is the Gaussian pulse, the pyramid and the square wave.

7.1.1 1D Gaussian pulse

The Gaussian pulse is a smooth function which its convection did not lead to numerical oscillations with any of the methods applied. The Upwind solution performed excessive diffusion in its solution and the BFECC method was capable to preserve the initial shape with good quality. The evaluation of this smooth function with the shock-capturing strategy in the BFECC method does not make much sense here, since numerical oscillations do not appear.

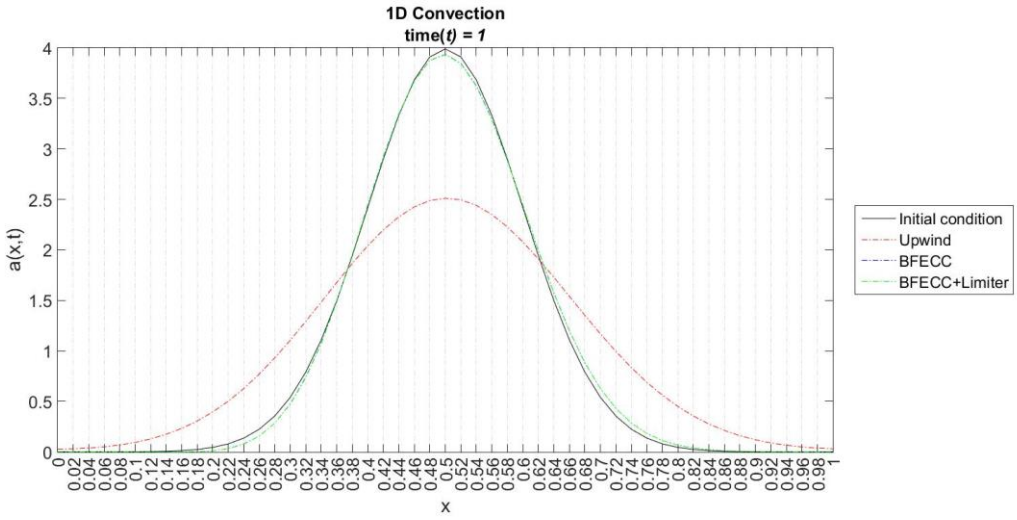


Figure 57. Correction of 1D Gaussian pulse advection with CFL = 0.5

It is verified that the solution of the BFECC method with shock-capturing strategy is similar to the classical BFECC solution.

CFL	Method	Overshooting	Undershooting
0.5	Upwind	-37.205%	0.733%
	BFECC	-1.395%	-0.044%
	BFECC + Shock-capturing	-1.425%	-0.006%

Table 17. 1D Gaussian pulse numerical oscillations correction with CFL = 0.5

For higher CFL values, the behaviour of the studied methods are almost the same.

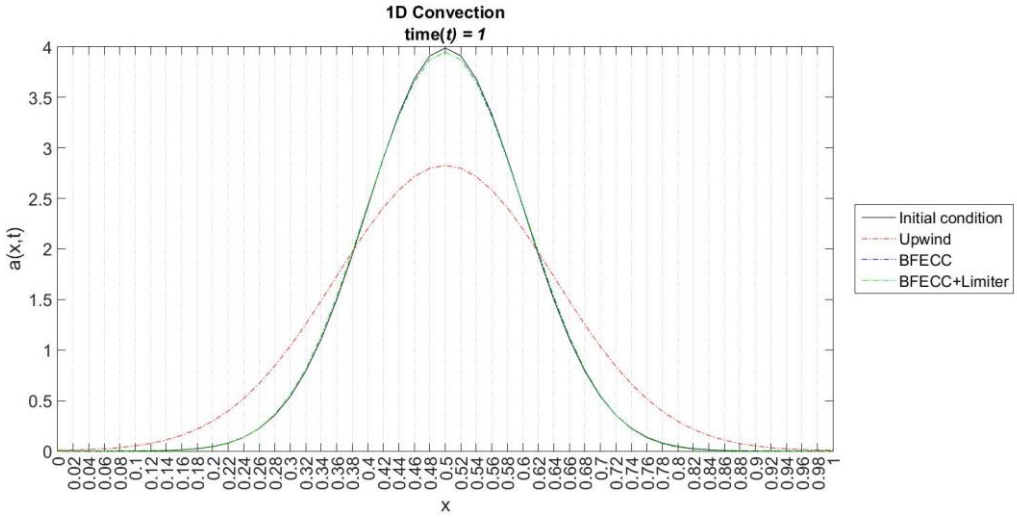


Figure 58. Correction of 1D Gaussian pulse advection with CFL = 1.0

The solution of the BFECC with shock-capturing strategy is closer to the BFECC solution for higher CFL values.

CFL	Method	Overshooting	Undershooting
1.0	Upwind	-29.489%	0.207%
	BFECC	-1.490%	-0.001%
	BFECC + Shock-capturing	-1.504%	-0.001%

Table 18. 1D Gaussian pulse numerical oscillations correction with CFL = 1.0

When CFL is equal to 2.0, it was seen that the Upwind method no longer shows diffusion but neither preserves the initial shape.

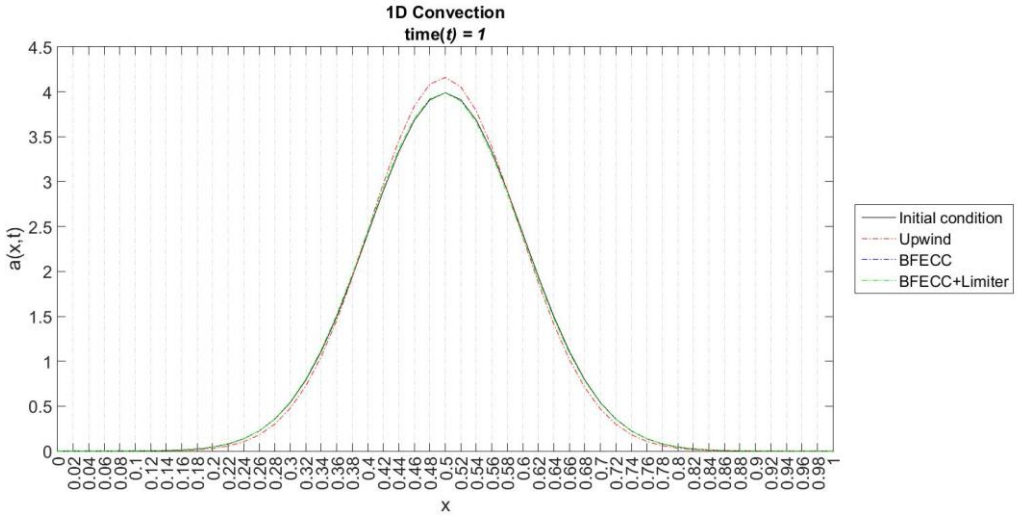


Figure 59. Correction of 1D Gaussian pulse advection with CFL = 2.0

The solution of the BFECC with the limiter and without it is even closer for CFL = 2.0.

CFL	Method	Overshooting	Undershooting
2.0	Upwind	4.367%	-0.000%
	BFECC	-0.021%	-0.000%
	BFECC + Shock-capturing	-0.022%	-0.000%

Table 19. 1D Gaussian pulse numerical oscillations correction with CFL = 2.0

7.1.2 1D Pyramid

Dealing with not so smooth functions the BFECC method with shock-capturing strategy seems to adapt better to the initial shape than the BFECC method.

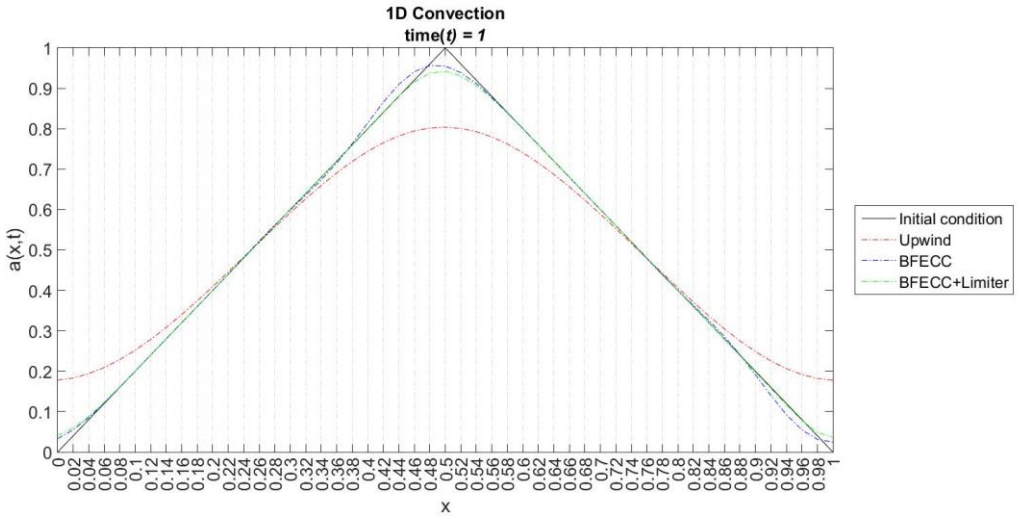


Figure 60. Correction of 1D Pyramid advection with CFL = 0.5

Even though, the overshooting and undershooting ratios are a bit larger in the improved method.

CFL	Method	Overshooting	Undershooting
0.5	Upwind	-19.698%	17.758%
	BFECC	-4.290%	2.504%
	BFECC + Shock-capturing	-5.768%	3.771%

Table 20. 1D Pyramid numerical oscillations correction with CFL = 0.5

There are almost no noticeable changes in the solution of the BFECC methods when increasing the CFL number, so they seem to be quite stable. It was seen that the Upwind method performs better results when increasing the CFL number, and in this case it can be corroborated.

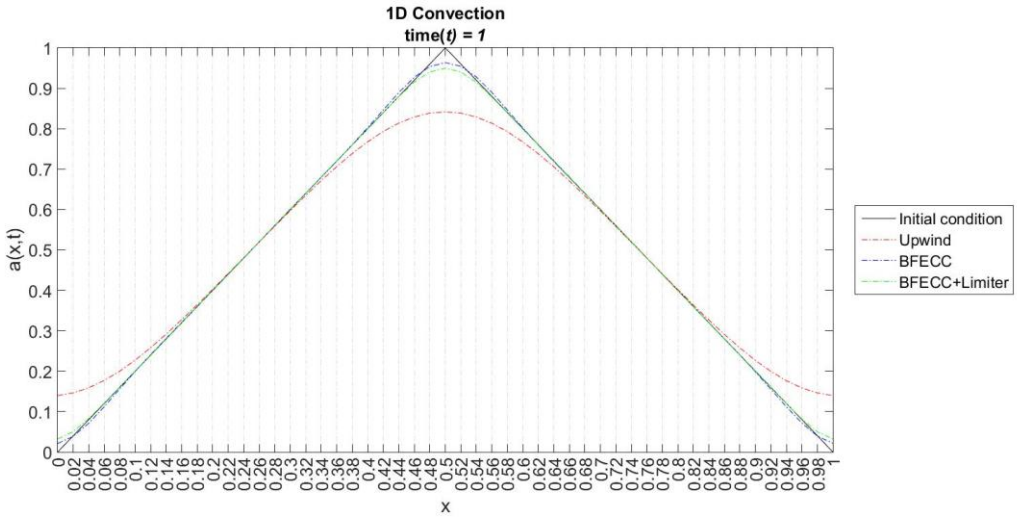


Figure 61. Correction of 1D Pyramid advection with CFL = 1.0

As in the previous case, the BFECC method with the shock-capturing strategy seems to slightly diffuse a little bit more than the BFECC method.

CFL	Method	Overshooting	Undershooting
1.0	Upwind	-15.989%	13.990%
	BFECC	-3.876%	1.905%
	BFECC + Shock-capturing	-5.257%	2.981%

Table 21. 1D Pyramid numerical oscillations correction with CFL = 1.0

In the case of larger CFL, the three compared methods seem to behave a bit better than before.

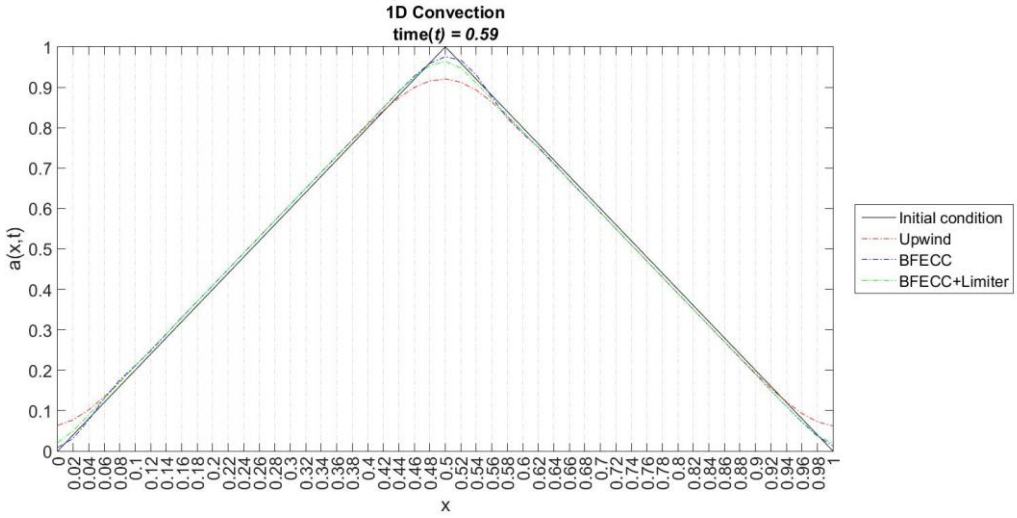


Figure 62. Correction of 1D Pyramid advection with CFL = 2.0

Although the solution of the BFECC method with shock-capturing strategy seems to be more diffused than in the BFECC solution, the shape of the function is better preserved and there are no point exceeding the initial function.

CFL	Method	Overshooting	Undershooting
2.0	Upwind	-8.043%	6.200%
	BFECC	-2.405%	0.991%
	BFECC + Shock-capturing	-3.665%	1.744%

Table 22. 1D Pyramid numerical oscillations correction with CFL = 2.0

7.1.3 1D Square wave

The case in which numerical oscillation appeared was the convection of a non-smooth function. Is in this case where the implementation of the shock-capturing strategy in BFECC method is powerful. Dealing with the convection of a square wave, the Upwind method is not capable at all to maintain the initial shape, but it also did not perform numerical oscillations. On the other hand, the BFECC method is capable of preserving in a better way the initial shape, but it performs numerical oscillations due to the sharp fronts of the function. It can be seen that the implementation of the shock-capturing strategy in the BFECC method leads to a much better solution. Particularly, the preserved shaped is almost the same than the simplest BFECC method and it is able to eliminate the spurious oscillations that where appearing.

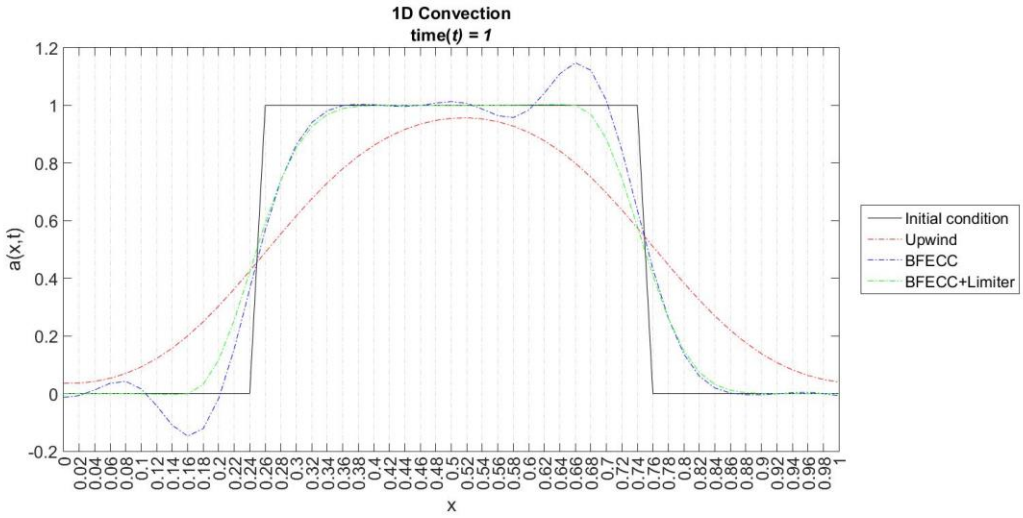


Figure 63. Correction of 1D Square wave advection with CFL = 0.5

Analysing the overshooting and undershooting ratios, it is confirmed that the shock-capturing strategy applied in the BFECC method can delete the large numerical oscillations that where performing the previous method (14.3% symmetrically on the top and on the bottom of the function).

CFL	Method	Overshooting	Undershooting
0.5	Upwind	-4.347%	3.605%
	BFECC	14.325%	-14.324%
	BFECC + Shock-capturing	0.276%	-0.295%

Table 23. 1D Square wave numerical oscillations correction with CFL = 0.5

The evaluation of the numerical test with larger CFL number has an improving effect in the Upwind method that has been seen before. In the BFECC solution still appear numerical oscillations and the shock-capturing strategy is skilful to eliminate them.

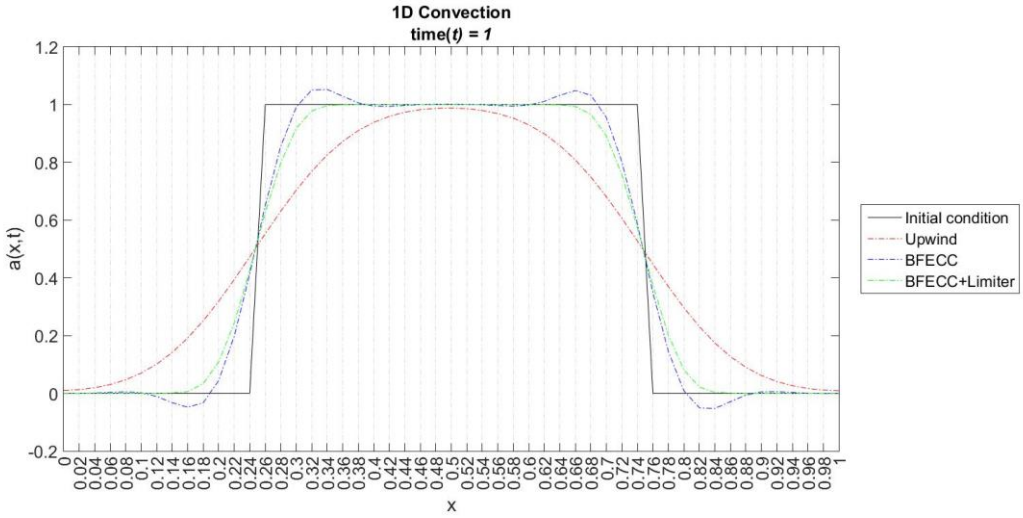


Figure 64. Correction of 1D Square wave advection with CFL = 1.0

The overshooting and undershooting ratios seem to be more favourable to the Upwind method than to the BFECC method, but looking at the picture one can realise that the shaped is better preserved with the BFECC method. It is remarkable that in this example the shock-capturing strategy has been able to completely eliminate all the numerical oscillations preserving the shape.

CFL	Method	Overshooting	Undershooting
1.0	Upwind	-1.220%	0.899%
	BFECC	5.671%	-5.671%
	BFECC + Shock-capturing	0.001%	-0.003%

Table 24. 1D Square wave numerical oscillations correction with CFL = 1.0

Finally, the case with the largest CFL number leads to pretty good results for the Upwind method and large numerical oscillations in the BFECC method as seen in previous chapters. The shock-capturing strategy applied in the BFECC method is also capable of eliminating the large numerical oscillations, that in this example grew up to almost 20% of the height of the function.

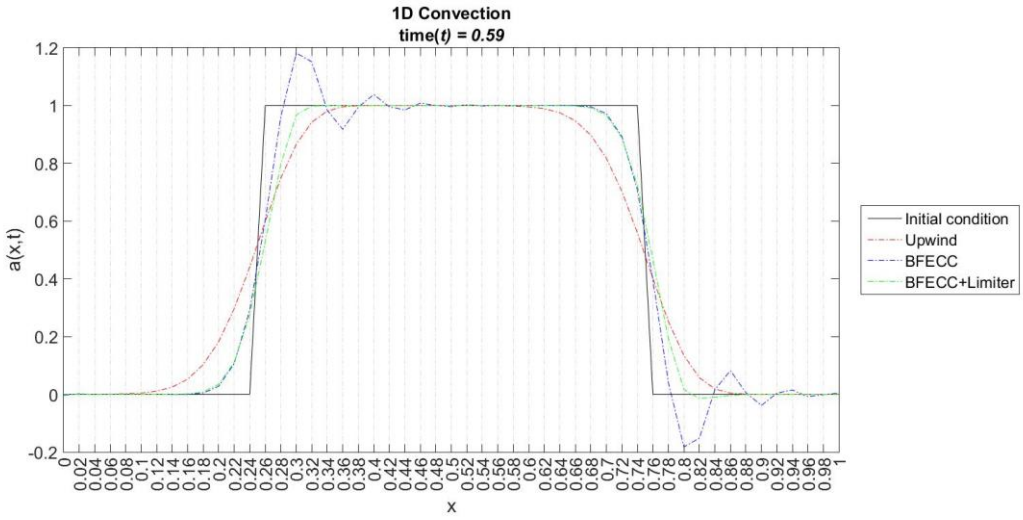


Figure 65. Correction of 1D Square wave advection with CFL = 2.0

The overshooting and undershooting ratios are again much better applying the shock-capturing strategy in the BFECC method than not applying it.

CFL	Method	Overshooting	Undershooting
2.0	Upwind	-0.000%	0.000%
	BFECC	19.155%	-19.155%
	BFECC + Shock-capturing	0.019%	-1.394%

Table 25. 1D Square wave numerical oscillations correction with CFL = 2.0

7.2 Kratos numerical test example in 2D with shock-capturing strategy

In this section the numerical test example performed in 5.3 is solved again with the application of the BFECC improved method presented in Chapter 6. The objective is to compare the results obtained and verify if the numerical oscillations that the simple BFECC was performing disappear with the shock-capturing strategy implemented in Kratos.

As a reminder, the problem that is pretended to solve is the movement of a heat focus either in time and space due to a velocity field imposed in the hole domain. The problem is physically governed by the pure convection equation, that is, a first order partial differential equation which is formed by the temperature vector and the velocity field.

The numerical test example is performed such that a heat focus is rotated for 10 seconds. The time discretization step used is 0.05 seconds, that leads to 200 time steps. The spatial discretization has been developed by the pre-processor GiD and read by Kratos as explained in 6.3.1.

The following figure shows the comparison between the previously BFECC method implemented in Kratos and the improvement of the BFECC method with the shock-capturing strategy.

On the left, there is the solution of the original BFECC method, that as seen in 5.3 performs numerical oscillations. On the right, the solution of the BFECC method with the shock-capturing strategy is presented. At the final time step equal to 10 seconds, it is clear that the spurious numerical oscillations have disappeared with the implementation of the limiting strategy. The original shape of the function is much better preserved that before, even though the initial condition corresponds to a highly non-smooth function. The method is capable of preserving the value in the centre of the heat focus, since the temperature in the focus is -1.0683 and with the simple BFECC method it was -1.2323. The temperature outside the heat focus is also well preserved, meaning that in the outer part of the heat focus the temperature maintains constant all over the domain and with the same value as in the initial condition.

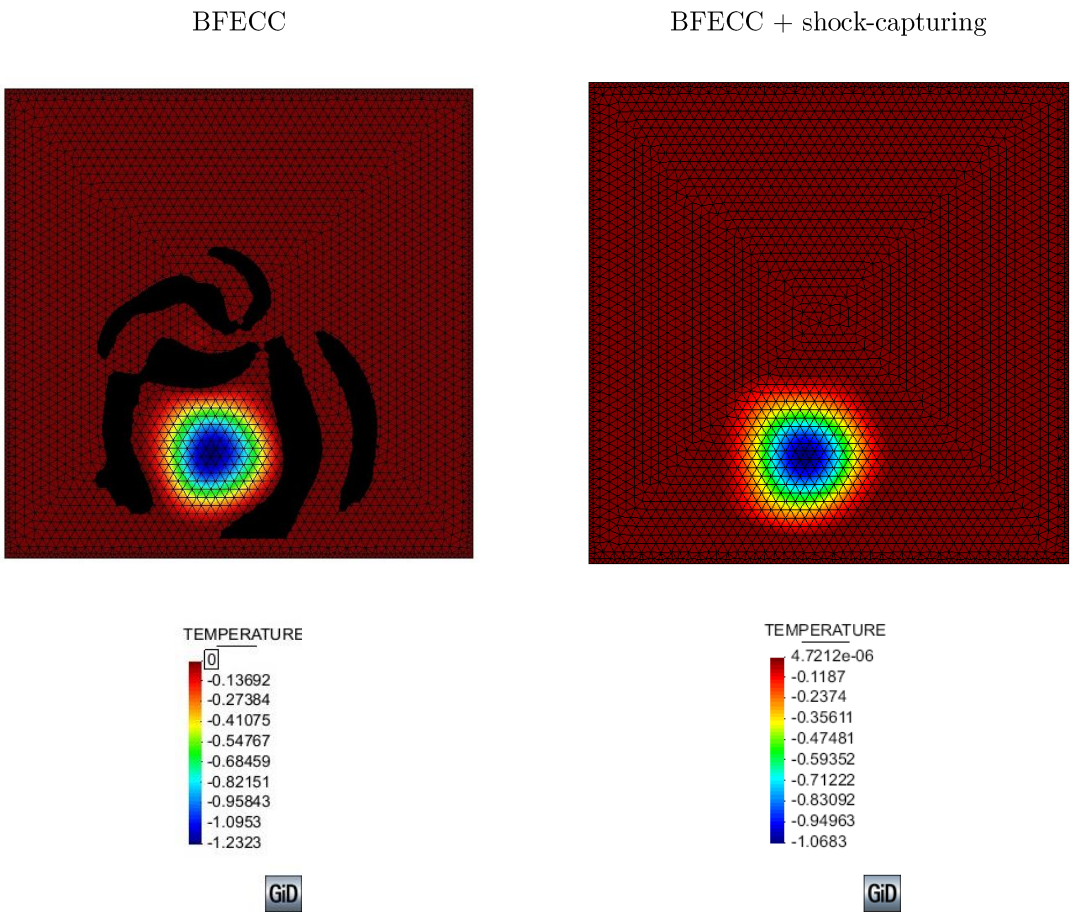


Figure 66. Comparison of final numerical test example' solution

The spurious numerical oscillations have disappeared, since there are no temperatures greater than 0 in any zone of the domain. Even so, the temperature in the centre of the heat focus is not exactly the same as in the initial condition. But, this phenomenon is due to the implicit diffusion effect of these methods, not because of the numerical oscillations.

	Time step	**Initial condition**	**BFECC + shock-capturing**	**Overshooting/ undershooting**
Minimal temperature	200	-1	-1.0683	6.83%
Maximal temperature	200	0	4.7212e-06	0.00%

Table 26. BFECC with shock-capturing strategy solution

In section 6.4 it has been presented an alternative way where to apply the shock-capturing strategy. As explained in that section, in Finite Elements codes it is usual to work with the element's barycentre, so it has been implemented the shock-capturing strategy in this points.

The figures below show the results when applying the shock-capturing strategy in the element's barycentre instead of applying it in the element's nodes.

At final step of the numerical test example, the application of the shock-capturing strategy at the element's barycentre does perform some numerical oscillations. This modification is better than the original BFECC method but is not capable of identifying all the numerical oscillations that may appear, as the shock-capturing strategy applied in the nodes does.

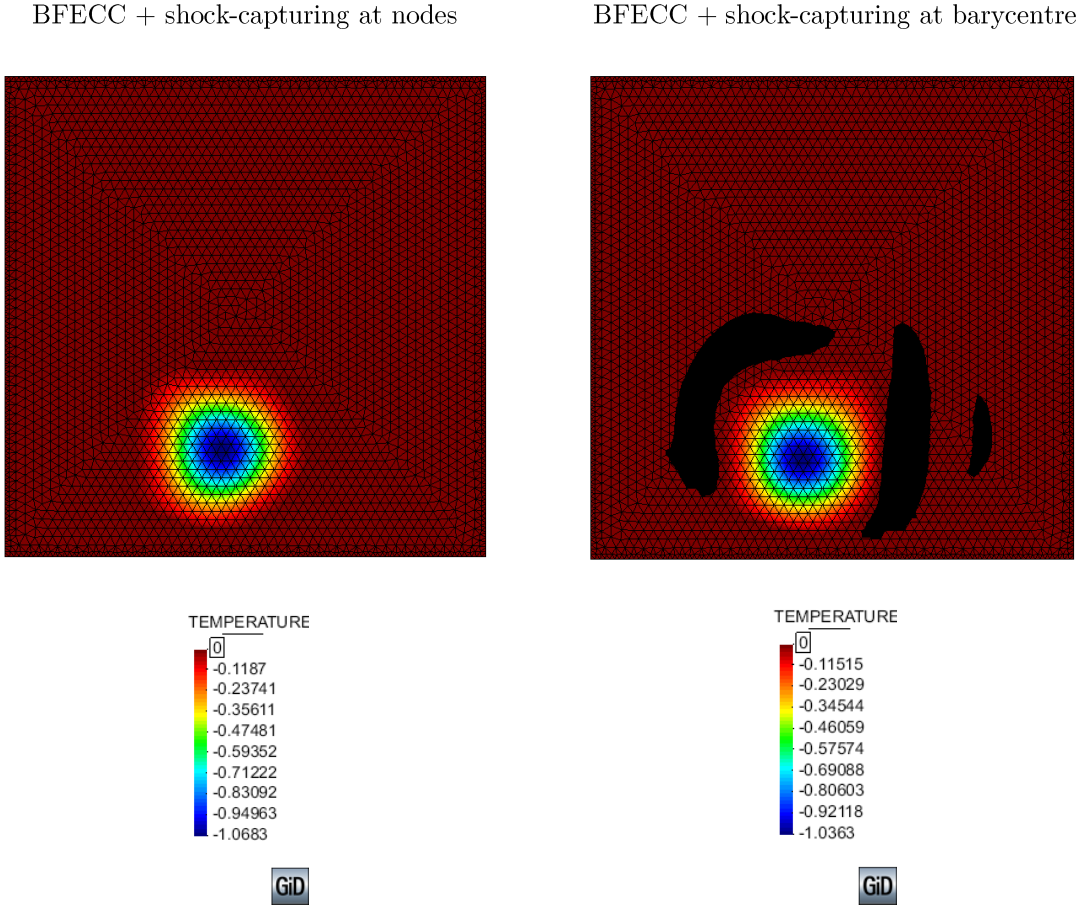


Figure 67. Comparison of final numerical test example' solution

Certainty, the solution in the centre of the heat focus is better when applying the shock-capturing strategy at the barycentre of the elements than the same strategy at the nodes. The solution at the centre of the heat focus is -1.0363, so the initial temperature of -1 is well preserved.

But the interesting thing here is whether these numerical oscillations that perform the application of the limiting strategy at the barycentre of the elements are large or not.

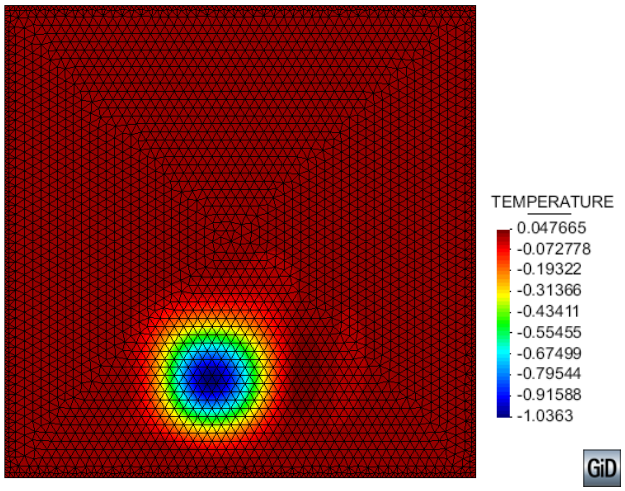


Figure 68. Numerical oscillations of shock-capturing at the barycentre

The numerical oscillations due to the application of the shock-capturing strategy at the element's barycentre seem to cover quite a vast area of the domain. Even so, it can be observed that these numerical oscillations are much smaller than without the application of any shock-capturing strategy.

	Time step	**Initial condition**	**BFECC + shock-capturing at barycentre**	**Overshooting/ undershooting**
Minimal temperature	200	-1	-1.0363	3.63%
Maximal temperature	200	0	0.0477	4.77%

Table 27. BFECC with shock-capturing strategy at the element's barycentre

CHAPTER 8

Conclusions

The thesis aims to implement a shock-capturing strategy in the BFECC method for Kratos in order to eliminate the spurious numerical oscillations that may occasionally appear in the problem's solution. To do so, the procedure that has been followed in this work goes from properly understanding the actual weaknesses of the solver to the validation of the computer implementation of the shock-capturing strategy.

Firstly, the BFECC method has been analysed under different conditions in order to observe when this method produces false numerical oscillations. Many test examples were run varying the initial condition function (from smooth to non-smooth functions with sharp fronts) and the CFL number, which are the two main decisive characteristics related with the behaviour of the solution. The test examples were performed using Matlab in one and two dimensions. The **BFECC method showed to be stable for smooth functions** regardless of the CFL number. When dealing with **rough functions with sharp fronts the BFECC method present numerical oscillations** around the irregular zone of the function. Increasing the CFL number the numerical oscillations seemed to slightly diffuse.

Then, the BFECC method has been analysed in Kratos. The test example ran in Kratos showed important numerical oscillations when the problem of the heat focus was solved using the BFECC method. Specifically, the solution performed a peak diffusion inside the heat focus of 23,23% and an undershooting outside of it of 13,22%.

Secondly, the computer implementation of the shock-capturing strategy has been made. Analysing the solutions of the BFECC method with the shock-capturing strategy in the one-dimensional test examples performed one can conclude that:

- For solving **smooth functions, the solution is as good as the BFECC solution** without the limiter.
- When dealing with **non-smooth functions the shock-capturing strategy is able to eliminate the spurious oscillations** and improve the BFECC solution.

In Kratos, the implementation has been made in two ways. On the one hand, the shock-capturing strategy has been applied at the elements' nodes, which is the original procedure of the limiting scheme. Moreover, it has been explored the consistency of applying the shock-capturing strategy at the barycentre of the elements, for computational reasons related to the structure of the solver's bases, further explained in 6.4. The verification and validation of the shock-capturing implementation in the BFECC method for Kratos have been done performing the rotating heat focus test example in two-dimension with the two approaches abovementioned. The following conclusions have been observed:

- When applying the shock-capturing strategy at the **elements' nodes** the solution seems to **display some diffusion**. In the Kratos test example, the solution inside the heat focus diffuses up to 6,83%, due to the intrinsic diffusive effect of these methods. Nevertheless, the method **totally eliminates the spurious numerical oscillations** that appeared in the solution of the BFECC method.
- When the shock-capturing is applied at the **elements' barycentre** a **few numerical oscillations come to light**. Even though, **the diffusion effect is reduced**. The Kratos test example solved following this approach showed a diffusion inside the heat focus of 3,63% and some numerical oscillations, up to 4,77% of undershooting, outside of it.

Conclusively, the objectives pursued in this thesis have been achieved. The shock-capturing strategy have been able to eliminate the spurious numerical oscillations performed by the original BFECC method. Nevertheless, the intrinsic diffusive effect of the BFECC method have slightly grown. To solve that, another way of implementing the shock-capturing strategy was studied, that is applying the limiting scheme at the elements' barycentre. With this approach, the numerical oscillation do not completely disappear, even though they are strongly reduced, but the diffusive effect is diminished. Moreover, applying the shock-capturing strategy at the elements' barycentre could be very useful for future improvements of the solver, considering that finite element bases usually refer to the barycentre of the elements in the mesh. Therefore, the thesis present two options for improving the original BFECC method with its strengths and weaknesses.

APPENDIX A. Matlab codes for solving numerical test examples

The current appendix contains the Matlab codes that have been used to perform the numerical test examples in the comparison analysis of different resolution methods in Chapter 5 and Chapter 7.

A.1. 1DConvection.m

This routine solves a one-dimensional advection problem using different methods on order to analyse the differences in the solutions performed by each method. The initial condition can be chosen out of three options corresponding to three different levels of smoothness of the function.

```
%    1DConvection.m
%    Numerical solution of the 1D linear advection equation u_t+cu_x=0
%    by finite difference methods on the domain 0<x<1, with periodic
%    boundary conditions and different initial condition functions.
%    The implemented methods are (1) Upwind, (2) BFECC and (3) BFECC
%    with minmod limiter.

%    06/2016 by Pau Vilar

clc; clear all;
```

SELECT NUMERICAL PARAMETERS

```
% Space parameters
L = 1;  %Length of grid
dx = 0.02;  %Grid spacing
fprintf('Grid spacing: dx = %.2f\n',dx);
nx = (1/dx)+1;  %Number of grid points
fprintf('Number of grid points: nx = %.0f\n',nx);
x = 0:dx:L; %Specifying the grid points

% Time parameters
tf = 1; %Final time
dt = 0.01;  %Width of each time step
fprintf('Width of each time step: dt = %.2f\n',dt);
nt = round(tf/dt);  %Number of time steps
fprintf('Number of time steps: nt = %.0f\n',nt);

% Velocity and Courant Number
c = 2.054; %Wave speed
fprintf('Wave speed = %.2f\n',c);
CFL = c*dt/dx; %Courant Number
```

```
fprintf('CFL = %.3f\n\n',CFL);
coeff = (c*dt)/(2*dx); %Coefficient for finite difference schemes
```

INITIAL CONDITION

```
a = zeros(1,nx);    %Preallocating initial condition
f_initial = menu('Choose the initial 1D function:','Square wave',...
    'Pyramid','Gaussian pulse');
if(f_initial == 1)
    a = f_piecewise(x);
elseif (f_initial == 2)
    a = f_pyramid(x);
elseif (f_initial == 3)
    a = normpdf(x,0.5,0.1);
end

% Initializing and prellocating vectors
a_up = a;
a_b = a;
astar_b = a;
anstar_b = a;
anhat_b = a;
a_bl = a;
astar_bl = a;
anstar_bl = a;
vnhat = a;
vstar = a;
anhat_bl = a;
e_1 = zeros(size(a));
e_2 = zeros(size(a));
e_1hat = zeros(size(a));
```

PERIODIC BOUNDARY CONDITIONS

```
io(1:nx) = 1:nx; %io = i
ip(1:(nx-1)) = 2:nx; ip(nx)=1; %ip = i+1 with periodic B.C.
im(2:nx) = 1:(nx-1); im(1)=nx; %im = i-1 with periodic B.C.
```

MAIN LOOP OVER DESIRED NUMBER OF TIME STEPS

```
for it = 1:nt
    % Initializing vectors at N time step
    an_up = a_up;
    an_b = a_b;
    an_bl = a_bl;

    % Plotting results
    fig = figure(1);
    set (fig, 'Units', 'normalized', 'Position', [0.05,0.15,0.9,0.7]);
```

```
    h = plot(x,a,'-k',x,a_up,'-.r',x,a_b,'-.b',x,a_bl,'-.g');
    legend('Initial condition','Upwind','BFECC','BFECC+Limiter',...
        'Location','eastoutside','Orientation','vertical');
    axis 'auto y'
    title({'1-D Convection';['time(\itt) = ',num2str(dt*it)]})
    xlabel('x'); ylabel('a(x,t)');
    ax = gca;
    ax.XGrid = 'on';
    ax.YGrid = 'off';
    ax.XTick = 0:dx:L;
    ax.GridLineStyle = '-.';
    ax.XTickLabelRotation = 45;
    drawnow;
    refreshdata(h)

    % Solution for each method
    % Upwind %
    a_up(io) = an_up(io) - coeff*(an_up(io)-an_up(im));

    % BFECC %
    astar_b(io) = an_b(io)-coeff*(an_b(io)-an_b(im));%Step1
    anstar_b(io) = astar_b(io)+coeff*(astar_b(ip)-astar_b(io));%Step2
    anhat_b(io)=0.5*(3*an_b(io)-anstar_b(io));%Step3
    a_b(io)=anhat_b(io)-coeff*(anhat_b(io)-anhat_b(im));%Step4

    % BFECC with limiting %
    %Step 1
    astar_bl(io)=an_bl(io)-coeff*(an_bl(io)-an_bl(im));%F with an
    %Step 2
    anstar_bl(io)=astar_bl(io)+coeff*(astar_bl(ip)-astar_bl(io));%B with astar
    %Step 3
    e_1(io)=0.5*(an_bl(io)-anstar_bl(io));%error1
    vnhat(io)=an_bl(io)+e_1(io);%an+e1
    vstar(io)=vnhat(io)-coeff*(vnhat(io)-vnhat(im));%F with vnhat
    %Step 4
    e_2(io)=an_bl(io)-((vstar(io)+coeff*(vstar(ip)-vstar(io))+e_1(io)));%error2
    %Step 5
    e_1hat(io) = e_1(io);
    for ii=2:nx-1
        if(abs(e_2(ii)) > abs(e_1(ii)))%identifying shocks
            e_1hat(ii-1) = minmod(e_1(ii),e_1hat(ii-1));%adjacent point before
            e_1hat(ii+1) = minmod(e_1(ii),e_1hat(ii+1));%adjacent point after
        end
    end
    %Step 6
    anhat_bl(io)=an_bl(io)+e_1hat(io);%corrected solution at N
    a_bl(io)=anhat_bl(io)-coeff*(anhat_bl(io)-anhat_bl(im));%Forward
end
```

RESULTS

```
% Error max value
t_up = ((max(a_up)-max(a))/max(a))*100;
fprintf('Overshooting of Upwind method = %.3f%%\n',t_up);
t_b = ((max(a_b)-max(a))/max(a))*100;
fprintf('Overshooting of BFECC method = %.3f%%\n',t_b);
t_bl = ((max(a_bl)-max(a))/max(a))*100;
fprintf('Overshooting of BFECC with limiter method = %.3f%%\n\n',t_bl);

% Error min value
b_up = (min(a_up)-min(a))/max(a)*100;
fprintf('Undershooting of Upwind method = %.3f%%\n',b_up);
b_b = (min(a_b)-min(a))/max(a)*100;
fprintf('Undershooting of BFECC method = %.3f%%\n',b_b);
b_bl = (min(a_bl)-min(a))/max(a)*100;
fprintf('Undershooting of BFECC with limiter method = %.3f%%\n',b_bl);
```

A.2. f_piecewise()

The function corresponds to a non-smooth shape of a square wave.

```
function y = f_piecewise(X)% Piecewise function

y = zeros(size(X));
region0 = (X <= 0.25) & (X >= 0.75);
y(region0) = 0;

region1 = (X > 0.25) & (X < 0.75);
y(region1) = 1;

end
```

A.3. f_pyramid()

This function performs a pyramidal shape, which is not smooth but less rough than the square wave.

```
function y = f_pyramid(X)% Pyramid function

y = zeros(size(X));
y = 1-abs(2*X-1);

end
```

A.4. 2DConvection.m

This Matlab routine solves a two-dimensional advection problem using different methods on order to analyse the differences in the solutions performed by each method. The initial condition can be chosen out of three options corresponding to three different levels of smoothness of the function.

```
%    2DConvection.m
%    Numerical solution of the 2D linear advection equation u_t+cu_x+cu_y=0
%    by finite difference methods on the domain [0,1]x[0,1], with periodic
%    boundary conditions and different initial condition functions.
%    The implemented methods are (1) Upwind, (2) BFECC and (3) BFECC
%    with minmod limiter.

%    06/2016 by Pau Vilar

clc; clear all;
```

SELECT NUMERICAL PARAMETERS

```
initial = menu('Choose the initial condition:','Square wave',...
    'Gaussian pulse','Cosine hump');
method = menu('Choose a numerical method:','Upwind','BFECC', ...
    'BFECC+Limiter');

% Space aprameters
Lx = 2;                     %Length of grid
Ly = 2;
dx = 0.04;
dy = 0.04;
nx = (Lx/dx)+1;             %Number of steps in space(x)
ny = (Ly/dy)+1;             %Number of steps in space(y)
x=0:dx:Lx;                  %Range of x(0,2) and specifying the grid points
y=0:dy:Ly;                  %Range of y(0,2) and specifying the grid points

% Time parameters
nt=150;                     %Number of time steps
dt=0.01;                    %Width of each time step

% Velocity field
vx = 3;                     %Velocity in x direction
vy = 3;                     %Velocity in y direction
Cx =(dt*vx)/(dx*2);
Cy =(dt*vy)/(dx*2);

% Preallocating vectors
u=zeros(nx,ny);
un=zeros(nx,ny);
ustar=zeros(nx,ny);
```

```
unstar=zeros(nx,ny);
unhat=zeros(nx,ny);
```

INITIAL CONDITION

```
if(initial == 1)% Square wave
    for i=1:nx
        for j=1:ny
            if ((0.5<=y(j))&&(y(j)<=1)&&(0.5<=x(i))&&(x(i)<=1))
                u(i,j)=1;
            else
                u(i,j)=0;
            end
        end
    end
elseif(initial == 2)% Gaussian pulse
    for i=1:nx
        for j=1:ny
            u(i,j) = gaussian2D(x(i),y(j));
        end
    end
elseif(initial == 3)% Cosine hump
    for i=1:nx
        for j=1:ny
            u(i,j) = cosHump2D(x(i),y(j));
        end
    end
end

% Ploting the initial condition
u0=u;
fig = figure(1);
set (fig, 'Units', 'normalized', 'Position', [0.05,0.15,0.9,0.7]);
subplot(1,2,1), surf(x,y,u0);
axis([0 2 0 2 0 1.5])
title('2-D Convection: Initial condition')
xlabel('x')
ylabel('y')
zlabel('u(x,y)')
colorbar
```

PERIODIC BOUNDARY CONDITIONS

```
ip(1:(nx-1)) = 2:nx; ip(nx)=1; % ip = i+1 with periodic B.C.
im(2:nx) = 1:(nx-1); im(1)=nx; % im = i-1 with periodic B.C.
i(1:nx) = 1:nx;
jp(1:(ny-1)) = 2:ny; jp(ny)=1; % jp = j+1 with periodic B.C.
jm(2:ny) = 1:(ny-1); jm(1)=ny; % jm = j-1 with periodic B.C.
j(1:ny) = 1:ny;
```

MAIN LOOP OVER DESIRED NUMBER OF TIME STEPS

```
for it=0:nt
    un=u;
    subplot(1,2,2), h=surf(x,y,u);       %plotting the velocity field
    axis([0 2 0 2 0 1.5])
    title({'2-D Convection';...
        'Transport property vector field {\bfu}=(u_x,u_y)';...
        ['time(\itt) = ',num2str(dt*it)]})
    xlabel('x')
    ylabel('y')
    zlabel('u(x,y)')
    colorbar
    drawnow;
    refreshdata(h)

    if(method == 1)% Upwind %
        u(i,j)=un(i,j)-Cx*(un(i,j)-un(im,j))-Cy*(un(i,j)-un(i,jm));

    elseif(method == 2)% BFECC %
        ustar(i,j) = un(i,j)-Cx*(un(i,j)-un(im,j))-Cy*(un(i,j)-un(i,jm));
        unstar(i,j) = ustar(i,j)+Cx*(ustar(ip,j)-ustar(i,j))+...
            Cy*(ustar(i,jp)-ustar(i,j));
        unhat(i,j) = 1.5*un(i,j)-0.5*unstar(i,j);
        u(i,j) = unhat(i,j)-Cx*(unhat(i,j)-unhat(im,j))-Cy*...
            (unhat(i,j)-unhat(i,jm));

    elseif(method == 3)% BFECC + Limiter %
        %Step 1: Forward with un
        ustar(i,j) = un(i,j)-Cx*(un(i,j)-un(im,j))-Cy*(un(i,j)-un(i,jm));
        %Step 2: Backward with ustar
        unstar(i,j) = ustar(i,j)+Cx*(ustar(ip,j)-ustar(i,j))+...
            Cy*(ustar(i,jp)-ustar(i,j));
        %Step 3: Error 1 and Forward
        e_1(i,j) = 0.5*(un(i,j)-unstar(i,j));
        vnhat(i,j) = un(i,j)+e_1(i,j);
        vstar(i,j) = vnhat(i,j)-Cx*(vnhat(i,j)-vnhat(im,j))-...
            Cy*(vnhat(i,j)-vnhat(i,jm));%Forward
        %Step 4: Error 2
        e_2(i,j)=un(i,j)-((vstar(i,j)+Cx*(vstar(ip,j)-vstar(i,j))+...
            Cy*(vstar(i,jp)-vstar(i,j)))+e_1(i,j));%Backward
        %Step 5: Limiter
        e_1hat(i,j) = e_1(i,j);
        for ii=2:nx-1
            for jj=2:ny-1
                if(abs(e_2(ii,jj)) > abs(e_1(ii,jj)))
                    e_1hat(ii-1,jj) = -minmod(e_1(ii,jj),e_1hat(ii-1,jj));
                    e_1hat(ii,jj-1) = -minmod(e_1(ii,jj),e_1hat(ii,jj-1));
                    e_1hat(ii+1,jj) = -minmod(e_1(ii,jj),e_1hat(ii+1,jj));
                    e_1hat(ii,jj+1) = -minmod(e_1(ii,jj),e_1hat(ii,jj+1));
                end
```

```
            end
        end
        %Step 6: Forward with modified solution
        unhat(i,j) = un(i,j)+e_1hat(i,j);
        u(i,j) = unhat(i,j)-Cx*(unhat(i,j)-unhat(im,j))-...
            Cy*(unhat(i,j)-unhat(i,jm));%Forward
    end
end
```

A.5. gaussian2D()

The function performs a Gaussian smooth shape.

```
function z=gaussian2D(x,y)
% 2D Gaussian function of height 1, centred on (cenx,ceny)
% which becomes zero rad away from the centre.

% cenx, ceny centre for Gaussian
cenx=1/2;
ceny=1/2;
% rad is Gaussian 'radius'
rad=0.1;
z=0.0;
d=sqrt((x-cenx)^2+(y-ceny)^2);% distance from centre
if (d < rad)
    z=exp(-0.01*d^2);
end
```

A.6. cosHump2D()

A smoother shape of a cosine hump is performed with this function.

```
function z = cosHump2D(x,y)

cenx = 1;
ceny = 1;
rad = 1;
d = min(sqrt((x-cenx)^2+(y-ceny)^2),rad)/rad;
z = 0.0;
if (d < rad)
    z = (1/4)*(1+cos(pi()*d));
end
```

A.7. minmod()

The *minmod* function is part of the limiting process of the shock-capturing strategy and evaluates the minimum of the modulus between two scalars if they have the same sign, otherwise the function returns a zero value.

```
function f = minmod(x,y)

if(x > 0 && y > 0)
    f = min(x,y);
elseif(x < 0 && y < 0)
    f = max(x,y);
else
    f = 0;
end

end
```

References

[1] Batchelor, G. K. (1967). *An Introduction to Fluid Dynamics*. Cambridge University Press.

[2] Eckert, M. (2006). *The Dawn of Fluid Dynamics: A Discipline Between Science and Technology*.

[3] Anderson, J. D. (1995). *Computational Fluid Dynamics: The Basics With Applications. Science/Engineering/Math*. McGraw-Hill Science.

[4] Patankar, S. V. (1980). *Numerical Heat Transfer and Fluid Flow. Hemisphere Series on Computational Methods in Mechanics and Thermal Science*. Taylor & Francis.

[5] GiD, The personal pre and post processor (www.gidhome.com), CIMNE.

[6] Parker, C. B. (1994). *Encyclopaedia of Physics (2nd ed.)*. McGraw Hill.

[7] Hoffman, J.D. (1993), *Numerical methods for engineers and scientists.* New York.

[8] CIMNE, International Centre for Numerical Methods in Engineering (www.cimne.com).

[9] Evans, L. C. (1998), *Partial Differential Equations*. American Mathematical Society.

[10] Washek, F. P. (2012), *The Divergence Theorem and Sets of Finite Perimeter*. University of California.

[11] Gottlieb, D. and Tadmor, E. (1991), *The CFL condition for spectral approximations to hyperbolic initial-boundary value problems.* American Mathematical Society.

[12] Bourke, P. (1999), *Interpolation methods*. Sydney, Australia.

[13] Edelsbrunner, H. (2001), *Geometry and Topology for Mesh Generation*, Cambridge University Press.

[14] Frey, P. J. and George, P. L. (2000), *Mesh Generation: Application to Finite Elements*, Hermes Science.

[15] Thompson, J. F.; Warsi, Z. U. A. and Mastin, C. W. (1985), *Numerical Grid Generation: Foundations and Applications*, North-Holland, Elsevier.

[16] Patankar, S. V. (1980). *Numerical Heat Transfer and Fluid Flow*. New York: McGraw-Hill. p. 102.

[17] Ferziger, J. H. and Peric, M. (1996), *Computational Methods for Fluid Dynamics*. Springer.

[18] Cockburn, B.; Karniadakis, G. E. and Shu, C. W. (2000), *The development of discontinuous Galerkin methods. In: Discontinuous Galerkin Methods. Theory, Computation and Applications*, LNCSE 11, Springer, 3–50.

[19] Atkinson, K. A. (1989), *An Introduction to Numerical Analysis (2nd ed.)*, New York: John Wiley & Sons, p. 20.

[20] Courant, R.; Friedrichs, K. and Lewy, H. (1928), *On the partial difference equations of mathematical physics*, IBM Journal of Research and Development 11 (2): 215–234.

[21] Hu, L; Li, Y and Liu, T (2013), *A limiting strategy for the back and forth error compensation and correction method for solving advection equations*. Mathematics of Computation.

Consulted literature

Rezzolla, L. (2011), *Numerical Methods for the Solution of Partial Differential Equations*. Lecture Notes for the COMPSTAR School on Computational Astrophysics, Caen, France.

Kuzmin, D. (2010), *A Guide to Numerical Methods for Transport Equations*. Friedrich-Alexander-Universität, Erlangen-Nürnberg.

Shu, C. W. and Osher, S. (1988), *Efficient implementation of essentially nonoscillatory shock-capturing schemes*. J. Comput. Phys., Vol. 77, No. 2, pp. 439–471.

Shu, C. W. and Osher, S. (1989), *Efficient implementation of essentially nonoscillatory shock-capturing schemes. II*. J. Comput. Phys., Vol. 83, No. 1, pp. 32–78.

Tadmor, E. (1990), *Shock capturing by the spectral viscosity method*. Comput. Methods Appl. Mech. Engrg., Vol. 80, No. 1-3, pp. 197–208.

Brooks, A. and Hughes, T. (1982), *Streamline upwind/Petrov-Galerkin formulations for convection dominated flows with particular emphasis on the incompressible Navier-Stokes equations*. Comput. Methods Appl. Mech. Engrg., Vol. 32, No. 1-3, pp. 199–259.

Nazarenko, S. (2014), *Fluid Dynamics via Examples and Solutions*. CRC Press (Taylor & Francis group).

Girault, V. and Raviart, P.-A. (1979), *Finite element approximation of the navierstokes equations*, Lecture Notes in Mathematics, Berlin Springer Verlag.

Dupont, T. F. and Liu, Y. (2003), *Back and forth error compensation and correction methods for removing errors induced by uneven gradients of the level set function*. Journal of Computational Physics, vol. 190, no. 1, pp. 311–324.

Abgrall, R. (1996), *Numerical discretization of the rst-order Hamilton-Jacobi equation on triangular meshes*, Comm. Pure Appl. Math.

Bryson, S. and Levy, D. (2003), *High-Order Semi-Discrete Central-Upwind Schemes for Multi-Dimensional Hamilton-Jacobi Equations*. J. Comp. Phys., pp. 63–87.

Lentine, M.; Gretarsson, J. and Fedkiw, R. (2011), *An Unconditionally Stable Fully Conservative Semi-Lagrangian Method*, J. Comp. Phys.

Gugushvili, I. V. and Evstigneev, N. M. (2009), *Semi-Lagrangian Method for Advection Equation on GPU in Unstructured R^3 Mesh for Fluid Dynamics Application*. World Academy of Science, Engineering and Technology.

Kim, B.-M.; Liu, Y.-J.; Llamas I. and Rossignac, J. (2005), *FlowFixer: Using BFECC for Fluid Simulation.* Eurographics Workshop on Natural Phenomena.

Kurganov, A. and Tadmor, E. (2000), *New High-Resolution Semi-Discrete Central Schemes for Hamilton-Jacobi Equations.* J. Comput. Phys., pp. 720–742.

Osher, S. and Shu, C.-W. (1991), *High-order essentially nonoscillatory schemes for HamiltonJacobi equations.* SIAM J. Numer. Anal.

Strain, J. (1999), *Semi-Lagrangian methods for level set equations*, J. Comput. Phys., pp. 498–533.

Enright, D.; Losasso, F. and Fedkiw, R. (2005), *A fast and accurate semi-lagrangian particle level set method.* Computers and Structures, pp. 479–490.

Warming, R.F. and Beam, R.M. (1976), *Upwind second-order difference schemes and applications in aerodynamic flows.* AIAA J.

Gilat, A. and Subramaniam, V. (2014), *Numerical Methods for Engineers and Scientists, An Introduction with Applications using MATLAB.* Department of Mechanical Engineering, The Ohio State University.

Vesely, F. J. (1994), *Computational Physics: An Introduction.* Plenum, New York, USA.

Smith, G. D. (1986), *Numerical Solution of Partial Differential Equations: Finite Difference Methods.* Oxford Univesity Press, third edition.

Lynch, L. (2004), *Numerical Integration of Linear and Nonlinear Wave Equations.* Florida Atlantic University.

Mathews, J. H. and Fink, K. D. (1999), *Numerical Methods Using MATLAB.* Prentice-Hall, Inc.

Kuzmin, D. (2010), *A Guide to Numerical Methods for Transport Equations.* Friedrich-Alexander-Universität, Erlangen-Nürnberg.

De Sterck, H. and Ullrich, P. (2009), *Introduction to Computational PDEs.* Department of Applied Mathematics, University of Waterloo.

LeVeque, R. J. (2005), *Finite Difference Methods for Differential Equations.* University of Washington.

Fazio, R. and Jannelli, A. (2004), *Second Order Positive Schemes by means of Flux Limiters for the Advection Equation.* IAENG International Journal of Applied Mathematics.